LBD Functions

As discussed in the previous chapters, int() is a built-in function in Python that facilitates typecasting, wherein a given value is converted to an integer. Typecasting involves converting one data type to another.

Let’s illustrate this with an example:

(In Python)

```
num_float = 5.7

num_int = int(num_float)

print(num_int)
```

In this example, the int() function is applied to the floating-point number 5.7. The function essentially truncates the decimal part, transforming the number into an integer.

Similarly, we can use the complex() function to create complex numbers by providing both real and imaginary parts.

Let’s see an example:

```
complex_num = complex(2, 5)

print(complex_num)
```

complex(2, 5) creates a complex number with a real part of 2 and an imaginary part of 5 to form the complex number (2 + 5j).

This showcases how we can convert one data type into another by leveraging specific functions in Python.

Moving on, let's discuss another built-in function pow(), which is used for exponential power calculations. For example:

```
result = pow(2, 3)

print(result)
```

pow() function takes two arguments: 2 as the base and 3 as the exponent. The function calculates

, which is the same as 2 raised to the power of 3. The result is stored in the variable result, and when we print it, the output will be: 8

Next we have a round() function that rounds the number to the nearest integer. For example:

```
num_float = 3.78

rounded_num = round(num_float)

print(rounded_num)
```

Here, the original num_float was 3.78, and the round() function rounded it to the nearest integer, which is 4. The round() function is handy when you want to work with whole numbers instead of decimals.

Now, let's introduce the len() function, which is used to find the length of various data type. Let’s illustrate it with an example:

(In Python)

```
string_example = "Hello, World!"

length = len(string_example)

print(length)
```

In this example, len() is applied to the string "Hello, World!" to find its length. The result is stored in the variable length, and when we print it, the output will be the number of characters in the string, in this case, 13. The len() function considers all characters, including spaces, providing a comprehensive measure of the length of the string. This versatility extends to other data types, making len() a useful tool for various applications.

For instance, consider the following list example:

```
list_example = [1, 2, 3, 4, 5]

length_list = len(list_example)

print(f"The length of the list is: {length_list}")
```

Here, len() is applied to the list [1, 2, 3, 4, 5]. The result is stored in the variable length_list, and when we print it, the output will be:

```
The length of the list is: 5
```

Now, let's introduce the min() and max() functions, which are used to find the minimum and maximum values in an iterable, respectively. For example:

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

min_value = min(numbers)

max_value = max(numbers)

print(f"The minimum value is: {min_value}")

print(f"The maximum value is: {max_value}")
```

In this example, min() is applied to the list of numbers, finding the smallest value, and max() finds the largest. The results are stored in the variables min_value and max_value, respectively. When we print these values, the output will be:

```
The minimum value is: 1

The maximum value is: 9
```

Now, let's explore the sum() function, which calculates the sum of all elements in an iterable. For example:

```
a = (1, 2, 3, 4)

b = [1, 2, 3, 4, 4]

c = {1, 2, 3, 4, 4}

sum_a = sum(a)

sum_b = sum(b)

sum_c = sum(c)

print(f"The sum of tuple a is: {sum_a}")

print(f"The sum of list b is: {sum_b}")

print(f"The sum of set c is: {sum_c}")
```

In this example, sum() is applied to a tuple a, a list b, and a set c. The results are stored in the variables sum_a, sum_b, and sum_c, respectively. When we print these values, the output will be:

```
The sum of tuple a is: 10
The sum of list b is: 14
```

The sum of set c is: 10, where it is the sum of unique elements in set c. Note that the difference between the sums of lists and sets is that in sets, the output is the sum of unique elements. As 4 is mentioned twice in sets, Python takes it to be only once.

Now let’s discuss the count() function, which counts the occurrences of a specific element in an iterable. For example:

```
word_list = ["apple", "banana", "apple", "orange", "apple", "grape"]
count_apple = word_list.count("apple")
print(f"The count of 'apple' in the list is: {count_apple}")
```

In this example, count() is applied to the list word_list to count the occurrences of the word "apple." The result is stored in the variable count_apple, and when we print it, the output will be:

```
The count of 'apple' in the list is: 3
```

Let’s see another example:

```
# Example string
my_string = "Hello, World!"
# Count occurrences of the letter 'o'
count_o = my_string.count('o')
# Print the result
print("The character 'o' appears", count_o, "times in the string.")
```

The output will be:

```
The character 'o' appears 2 times in the string.
```

In this example, count_o will be assigned the value 2 because there are two occurrences of the letter 'o' in the string "Hello, World!". You can replace 'o' with any other character you want to count in the string.

Let's discuss how to convert a string to uppercase and lowercase in Python using the upper() and lower() methods:

The upper() method in Python is used to convert all characters in a string to uppercase and The lower() method in Python is used to convert all characters in a string to lowercase.

Let’s see an example:

```
my_string = "Hello, World!"

# Convert the string to lowercase

lowercase_string = my_string.lower()

# Convert the string to uppercase

uppercase_string = my_string.upper()

# Print the result

print(f"Original string: {my_string}")

print(f"Lowercase string: {lowercase_string}")

print(f"Uppercase string: {uppercase_string}")
```

The output is:

```
Original string: Hello, World!

Lowercase string: hello, world!

Uppercase string: HELLO, WORLD!
```

In both examples, we start with an original string (my_string), and then we use the upper() method to create an uppercase version (uppercase_string) and the lower() method to create a lowercase version (lowercase_string). The results are printed to show the conversion.

capitalize() is another method in Python used to capitalize the first character of a string. Here's an example:

```
# Example sentence

my_sentence = "python is a versatile programming language."

# Capitalize the first character of the sentence

capitalized_sentence = my_sentence.capitalize()

# Print the results
```

```
print(f"Original sentence: {my_sentence}")
print(f"Capitalized sentence: {capitalized_sentence}")
```

The output will be:

```
Original sentence: python is a versatile programming language.
Capitalized sentence: Python is a versatile programming language.
```

In this example, the capitalize() method is applied to the original sentence (my_sentence), resulting in a new string (capitalized_sentence) where the first character of the sentence is capitalized. This method is particularly useful when you want to ensure that the first character of a string is in uppercase, making it suitable for use as a sentence or title.

Now, suppose in the sentence ‘python is a versatile programming language’ and I want to convert the first character of each word to uppercase. To do this we will use the title() built-in function.

Let’s see how:

```
my_sentence = "python is a versatile programming language."
# Convert the sentence to title case
title_sentence = my_sentence.title()
# Print the results
print(f"Original sentence: {my_sentence}")
print(f"Titlecased sentence: {title_sentence}")
```

The output will be:

```
Original sentence: python is a versatile programming language.
Titlecased sentence: Python Is A Versatile Programming Language.
```

Now let’s say in the sentence "python is a versatile programming language." I want to replace ‘versatile’ with powerful, this too can be done by another python built-in function, which is:

```
replace(old_substring, new_substring, count)
```

where old_substring is the substring to be replaced, new_substring is the substring to replace occurrences of the old substring and count is an optional parameter specifying the maximum number of occurrences to replace. If not provided, all occurrences are replaced.

Let's go through an example to illustrate the usage:

(In Python)

```
original_string = "python is a versatile programming language."

# Replace 'versatile' with 'powerful'

modified_string = original_string.replace('versatile', 'powerful')

# Print the results

print(f"Original string: {original_string}")

print(f"Modified string: {modified_string}")
```

The results are printed, showing the original and modified sentences with the replacement.

```
Original string: python is a versatile programming language.

Modified string: python is a powerful programming language.
```

Let's suppose we want to find the index of the word "versatile" in the sentence ‘Python is a versatile programming language’.

(In Python)

```
sentence = "Python is a versatile programming language."

# Let's find the index of the word "versatile" in the sentence

index = sentence.find("versatile")

# Display the result

print(f"The word 'versatile' is found at index: {index}")
```

So the output will be:

```
The word 'versatile' is found at index: 12
```

Now let’s take another scenario where the same code snippet is used and only the sentence is changed to a longer sentence. I have changed the sentence to a pre-written one.

(In python)

sentence = "Python is a versatile programming language, known for its versatile capabilities in various domains, making it a versatile choice for developers working on diverse projects."

index = sentence.find("versatile")

# Display the result

print(f"The word 'versatile' is found at index: {index}")

The output is:

The word 'versatile' is found at index: 12

In both cases, the occurrence of 'versatile' is found at index: 12, which clearly indicates that the find() method is used to locate the index of the first occurrence of the specified word which is 'versatile' in this context in the given sentences.

So the syntax is :

index = string_to_search.find(substring_to_find, start_index, end_index)

Where string_to_search is the main text or string in which you want to search for the substring.

substring_to_find is the specific part of the text you are looking for.

start_index which is optional is the index from which the search should start. If not provided, it starts from the beginning.

end_index which is too optional is The index at which the search should end. If not provided, it searches until the end of the string.

If the specific word is not found, it returns -1.

let's modify the above code to include the substring_to_find, start_index, and end_index parameters in the find() method:

(In Python)

# Original Sentence

sentence = "Python is a versatile programming language, known for its versatile capabilities in various domains, making it a versatile choice for developers working on diverse projects."

```
# Using find() with substring_to_find, start_index, and end_index

substring_to_find = "versatile"

start_index = 15

end_index = 80 # Adjust this based on the desired range

# Finding the index of the first occurrence of the specified substring within the given range

index = sentence.find(substring_to_find, start_index, end_index)

# Display the result

print(f"The word '{substring_to_find}' is found at index: {index}")
```

In this modified example, the find() method is used with the specified substring_to_find ("versatile"), start_index (15), and end_index (80). The start_index indicates the position to start searching, and the end_index indicates the position to stop searching. Adjust the end_index based on the desired range for the search. So the output is

```
The word 'versatile' is found at index: 58
```

Which reflects the index of the first occurrence of the specified word within the specified range.

Moving on, next we have is split() function.

split() method in Python is used to split a string into a list of substrings based on a specified delimiter.

The syntax is:

```
list_of_substrings = string_to_split.split(separator, maxsplit)
```

string_to_split is the original string that you want to split.

“

separator” which is optional is the delimiter or character at which the string should be split. If not provided, it splits at whitespaces.

And maxsplit, which too is optional, is the maximum number of splits to perform. If not provided, there is no limit on the number of splits.

Let’s see it with an example:

(In Python)

```
sentence = "Python is a versatile programming language."
word_list = sentence.split()
print("Original Sentence:", sentence)
print("List of Words:", word_list)
```

In this example, split() is applied to the sentence string without specifying a custom delimiter. Therefore, it uses the default whitespace character as the delimiter.

The result is stored in the word_list variable.

The original sentence and the list of words are then displayed as follows:

```
Original Sentence: Python is a versatile programming language.
List of Words: ['Python', 'is', 'a', 'versatile', 'programming', 'language.']
```

So the result is a list of words where each word is an element in the list.

Now let’s look another example of splitting a date entered by the user.

(In Python)

```
date_input = input("Enter a date (dd-mm-yyyy): ")
# Splitting the input into day, month, and year
day, month, year = date_input.split('-')
# Displaying the result
print("Day:", day)
print("Month:", month)
print("Year:", year)
```

Let’s say the date entered is 1-01-2024

So the output is:

```
Day: 1
```

Month: 01

Year: 2024

Here the split('-') function is used to split the user input into three parts: day, month, and year.

The hyphen is the separator. It tells the program where to split the input.

The day variable holds the first part (day), the month variable holds the second part (month), and the year variable holds the third part (year).

Suppose we modify our above sentence a bit by adding a custom delimiter, such as a semicolon.

So our modified code looks like this:

(In Python)

```
sentence = "Python; is; a; versatile; programming; language."

word_list = sentence.split(';', maxsplit=2)

print("Original Sentence:", sentence)

print("List of Words:", word_list)
```

In this modified example, the split() method not only uses a semicolon as the custom delimiter but also incorporates the maxsplit=2 parameter. This means the string is split into at most 2 substrings based on the semicolon.

The output would be:

```
Original Sentence: Python; is; a; versatile; programming; language.

List of Words: ['Python', ' is', ' a; versatile; programming; language.']
```

The string is split at the first two semicolons, and the remaining part of the string is treated as a single substring. The resulting list includes three elements: 'Python', ' is', and ' a; versatile; programming; language.'.

Next, we have startswith() and endswith() functions.

Whereas the name suggests the startswith() function checks if a string starts with a specific set of characters. And endswith() function checks if a string ends with a specific set of characters.

Let’s see it with an example:

(In Python)

```
file_name = "document.txt"

if file_name.startswith("document"):

print("This is a document file.")

elif file_name.startswith("image"):

print("This is an image file.")

else:

print("Unknown file type.")
```

This code checks the beginning of the file name to identify what type of file it might be. It's like looking at the starting part of a book title to figure out the genre. If it starts with "document," it's identified as a document file; otherwise, it checks for "image." If neither condition is met, it labels it as an unknown file type.

So since the file_name variable holds the name of a file which is "document.txt."

Therefore the output will be:

```
This is a document file.
```

Now let’s see an example of endswith() function.

(In Python)

```
email = input('Enter the email: ')

if email.endswith(".com"):

print("This is a valid email.")

else:

print("Invalid email format.")
```

The code is checking if the email address has a common format where it ends with ".com." If it doesn't, it considers it an invalid email format.

Let’s say the user enters "mansi@gmail"

In this case, the email does not end with ".com," so it goes to the else: part.

The output will be:

```
Invalid email format.
```

In addition to this, let’s say we want to check if the string starts with any of the specified prefixes provided in a tuple.

Let’s understand this with an example:

```
# Define a tuple of valid prefixes
valid_prefixes = ("http://", "https://", "ftp://")
# User input: a website URL
url = input("Enter a website URL: ")
# Check if the URL starts with any of the valid prefixes
if url.startswith(valid_prefixes):
print("The URL is valid.")
else:
print("Invalid URL. Please include a valid prefix like 'http://', 'https://', or 'ftp://'.")
```

In this example a tuple named valid_prefixes contains the common URL prefixes like "http://", "https://", and "ftp://".

The url.startswith(valid_prefixes) checks if the entered URL starts with any of the valid prefixes.

If the condition is met, it prints "The URL is valid." Otherwise, it prints a message suggesting valid prefixes.

(in Python)

Let’s say that we run the code and we input https://www.example.com

Since the input URL "https://www.example.com" starts with "https://", it matches one of the valid prefixes.

So the output will be:

The URL is valid.

Similar to startswith(tuple), endswith(tuple) checks whether it ends with any of the specified suffixes in the tuple.

Now let’s imagine you have a word written on a whiteboard, but there are some extra spaces at the beginning or end of the word. The strip() function is like wiping away those extra spaces, making the word clean and neat.

For example:

```
word_with_spaces = " Python "

cleaned_word = word_with_spaces.strip()

print("Original Word:", word_with_spaces)

print("Cleaned Word:", cleaned_word)
```

In this example, we have a word with extra spaces at the beginning and end: " Python ". The strip() method is then applied to clean up the word.

So the output will be:

```
Original Word: Python

Cleaned Word: Python
```

Let’s see another example to get a clearer view of what this function does.

(In Python)

```
user_names = [" John", "Alice ", " Bob ", "Charlie"]

# Cleaning up user names using strip()

cleaned_names = [name.strip() for name in user_names]

# Displaying the cleaned names
```

```
print("Original Names:", user_names)
print("Cleaned Names:", cleaned_names)
```

Let’s say we have a list of user names as user_names and some names in it have extra spaces at the beginning or end. So we’ll use strip() function that removes any leading or trailing spaces from each name, making them uniform.

cleaned_names = [name.strip() for name in user_names] is a compact approach called a list comprehension which is a concise way to create lists in Python. It iterates through each name in the user_names list. For each name, name.strip() is applied, creating a new list called the cleaned_names with cleaned-up names.

A more detailed way to express the usage of this compact approach, known as the traditional for loop, The basic syntax for this would be:

```
cleaned_names = []
for name in user_names:
cleaned_names.append(name.strip())
```

So our code with the alternate approach looks like:

```
user_names = [" John", "Alice ", " Bob ", "Charlie"]
# Cleaning up user names using strip()
cleaned_names = []
for name in user_names:
cleaned_names.append(name.strip())
# Displaying the cleaned names
print("Original Names:", user_names)
print("Cleaned Names:", cleaned_names)
```

(Show this alternate code in the ppt, and output of the code in Python to be showed from the first code)

So the output will be:

```
Original Names: [' John', 'Alice ', ' Bob ', 'Charlie']
```

```
Cleaned Names: ['John', 'Alice', 'Bob', 'Charlie']
```

The cleaned names demonstrate uniformity, with all leading and trailing spaces removed. This ensures a consistent representation of the user names.

Moving on let's consider a case where you want to create a list of user input for favorite movies:

(In Python)

```
# Initialize an empty list for favorite movies

favorite_movies = []

# Ask the user to input their favorite movies using a loop

for i in range(3): # Let's assume the user will provide three favorite movies

movie = input("Enter your favorite movie: ")

favorite_movies.append(movie)

# Display the final list of favorite movies

print("Your Favorite Movies:", favorite_movies)
```

An empty list named favorite_movies is initialized.

Let’s assume the for loop runs three times to get input for three favorite movies from the user.

Inside the loop, the input() function is used to take the user's input, and append() adds each movie to the favorite_movies list.

The print("Your Favorite Movies:", favorite_movies) statement displays the final list of favorite movies entered by the user.

Now let’s run the code and create a list of favorite movies (in Python).

(Enter the movie name):

Harry Potter

The Hangover

Dumb and Dumber

So the output is:

Your Favorite Movies: ['Harry Potter', 'The Hangover', 'Dumb and Dumber']

Next, we have is extend(). Used to modify the list by adding elements from another list.

Let's consider a situation where you have a shopping list from two different family members:

```
# Shopping list from person A

list_a = ["Apples", "Milk", "Bread"]

print('First list: ',list_a)

First list: ['Apples', 'Milk', 'Bread']

# Shopping list from person B

list_b = ["Eggs", "Cheese", "Tomatoes"]

print('Second list: ',list_b)

Second list: ['Eggs', 'Cheese', 'Tomatoes']

# Combining the shopping lists using extend()

list_a.extend(list_b)

# Displaying the combined shopping list

print("Combined Shopping List:", list_a)

Combined Shopping List: ['Apples', 'Milk', 'Bread', 'Eggs', 'Cheese', 'Tomatoes']
```

In this, the extend() method modifies the original list_a by adding elements from list_b.

The result is directly reflected in list_a, which is then printed to show the combined shopping list.

Now instead of modifying the list_a directly, we can modify the code

(This code to be explained in ppt)

```
# Shopping list from person A

list_a = ["Apples", "Milk", "Bread"]
```

```
# Shopping list from person B

list_b = ["Eggs", "Cheese", "Tomatoes"]

# Combining the shopping lists using extend()

combined_list = list_a.copy() # Copy the first list to avoid modifying it directly

combined_list.extend(list_b)

# Displaying the combined shopping list

print("Combined Shopping List:", combined_list)

print(list_a)
```

where the combined_list = list_a.copy() creates a copy of list_a using copy() and assigns it to the variable combined_list. The extend() method is then applied to combined_list, adding elements from list_b to the end of the copied list.

Now Imagine you have a row of toy cars, and you want to add a new toy car right at the second position. The insert() function helps you do just that – it lets you insert something at a particular position in a list of things.

Let's use the toy cars example to demonstrate the insert() function in a simple code snippet:

```
# Initial list of toy cars

toy_cars = ["Red Car", "Blue Car", "Green Car"]

# Display the original list

print("Original Toy Cars:", toy_cars)

# New toy car to be added

new_toy_car = "Yellow Car"

# Insert the new toy car at the second position (index 1)

toy_cars.insert(1, new_toy_car)

# Display the updated list

print("Updated Toy Cars:", toy_cars)
```

In this example, we start with a list of toy cars: ["Red Car", "Blue Car", "Green Car"].

We have a new toy car, "Yellow Car", that we want to insert into the list.

We use the insert() function to add the new toy car to the second position (index 1).

Python, indexing starts from 0. So, 1 refers to the second position in the list.

new_toy_car is the new element that we want to insert into the list.

So, toy_cars.insert(1, new_toy_car) essentially means "insert the new_toy_car at index 1 (the second position) in the toy_cars list." This operation modifies the original list by adding the new toy car to the specified position.

So the output is:

```
Original Toy Cars: ['Red Car', 'Blue Car', 'Green Car']

Updated Toy Cars: ['Red Car', 'Yellow Car', 'Blue Car', 'Green Car']
```

Moving on let’s say we have a list of numbers = [5, 2, 8, 1, 3] and we want to reverse the order of elements in a list i.e we want numbers=[3, 1, 8, 2, 5] so this in python can be done using reverse().

Let’s see how

(In Python)

```
numbers = [5, 2, 8, 1, 3]

# Display the original list

print("Original Numbers:", numbers)

# reversing the list

numbers.reverse()

# Display the reversed list

print("Reverse order:", numbers)
```

So the output will be:

```
Original Numbers: [5, 2, 8, 1, 3]

Reverse order: [3, 1, 8, 2, 5]
```

Here numbers.reverse() is like flipping the list. It rearranges the numbers so that the last one becomes the first, the second-to-last becomes the second, and so on.

Let’s see another scenario where we want to sort the list of numbers. This can be done using sort() function which is used for arranging elements in either ascending or descending order in a list.

Let’s see how:

Using the same code used in the previous example and modifying a few things in it.

(Highlights are for my reference only)

```
numbers = [5, 2, 8, 1, 3]

# Display the original list

print("Original Numbers:", numbers)

# Sort the list in ascending order

numbers.sort()

# Display the sorted list

print("Sorted Numbers (Ascending):", numbers)
```

The output is:

```
Original Numbers: [5, 2, 8, 1, 3]

Sorted Numbers (Ascending): [1, 2, 3, 5, 8]
```

Now to print the list in descending order we’ll modify the code a bit

```
numbers = [5, 2, 8, 1, 3]

# Display the original list

print("Original Numbers:", numbers)

# Sort the list in descending order

numbers.sort(reverse=True)
```

```
# Display the sorted list in descending order
print("Sorted Numbers (Descending):", numbers)
```

The reverse=True part is a boolean parameter. When set to False (or not specified, as it is by default), the list is sorted in ascending order. However, in this case, we set it to True explicitly, which tells Python to sort the list in descending order.

After this line, the list becomes [8, 5, 3, 2, 1]. And so our output is:

```
Original Numbers: [5, 2, 8, 1, 3]
Sorted Numbers (Descending): [8, 5, 3, 2, 1]
```

Next, we have is remove() function. This method is used to remove the first occurrence of a specified element from a list. Here's an example:

```
# Create a list of fruits
fruits = ["apple", "banana", "orange", "apple", "grape"]
# Print the original list
print("Original list:", fruits)
# Remove the first occurrence of "apple" from the list
fruits.remove("apple")
# Print the modified list
print("List after removing 'apple':", fruits)
```

The remove() method looks for the first occurrence of the specified value ("apple" in this case) in the list and removes it. Since "apple" appears twice in the original list, only the first occurrence is removed.

So the output looks like:

```
Original list: ['apple', 'banana', 'orange', 'apple', 'grape']
List after removing 'apple': ['banana', 'orange', 'apple', 'grape']
```

Next, we have is index() method.

Suppose you have a tuple and you want to find the position (index) of a specific element in the tuple:

Let’s see this with a code illustration:

(In Python)

```
# Create a tuple of days of the week

days_of_week = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday")

# Search for the position of "Wednesday" in the tuple

searched_day = "Wednesday"

# Get the position of the day without checking

position = days_of_week.index(searched_day)

# Print the result

print(f"{searched_day} is at position {position + 1} in the tuple.")
```

In this code we want to find the position of "Wednesday" in our tuple, so we create a variable called searched_day and set it to the day we're looking for.

index() method is then used on our tuple to find the position or index of the searched_day. This position is then stored in a variable called position.

Finally, we printed a message saying which day we were looking for which is "Wednesday" and at what position it was found in the tuple. The + 1 is there because positions in programming usually start from 0, but in everyday language, we often start counting from 1.

So, when you run this code, it will output:

```
Wednesday is at position 3 in the tuple.
```

Next, we have any() function.

The any() function in Python is often used when you have an iterable (such as a list, tuple, or other collection), and you want to check if at least one element in the iterable evaluates to True.

Let’s understand this with an example:

```
# List of boolean values

bool_values = [False, False, True, False]
```

```
# Check if at least one value is True
is_any_true = any(bool_values)
# Print the result
if is_any_true:
print("At least one value is True.")
else:
print("No value is True.")
```

In this example:

bool_values is a list containing boolean values.

any(bool_values): Here the any() function checks if at least one value in the list is True.

is_any_true: Stores the result of the any() check.

The output of this code will be

```
At least one value is True
```

because there is a True value in the list. If all values were False, it would print "No value is True."

In addition to this, if suppose we want to check if all the values in the list are True, we'll check this using all().

Let's understand this with an example:

In this example, we'll demonstrate the use of both any() and all() methods to check conditions on the temperatures:

```
# List of temperatures
temperatures = [25, 28, 30, 32, 29]
# Check if at least one temperature is above 30 degrees Celsius using any()
is_above_threshold_any = any(temp > 30 for temp in temperatures)
# Check if all temperatures are above 20 degrees Celsius using all()
are_all_above = all(temp > 30 for temp in temperatures)
```

```
# Output the results
if is_above_threshold_any:
print("At least one temperature is above 30 degrees Celsius.")
else:
print("No temperature is above 30 degrees Celsius.")
if are_all_above:
print("All temperatures are above 30 degrees Celsius.")
else:
print("Not all temperatures are above 30 degrees Celsius.")
```

In this example, the any() function is used to check if at least one temperature in the list is above 30 degrees Celsius and all() function is used to check if all the temperature in the list is above 30 degrees Celsius.

The generator expression (temp > 30 for temp in temperatures) generates True or False for each temperature in the list based on whether it's greater than 30.

Let's break it down into simpler parts:

```
for temp in temperatures:
```

This part is like saying, "For each temperature in the list of temperatures..."

```
temp > 30:
```

This part is like asking a question about each individual temperature: "Is this specific temperature greater than 30 degrees Celsius?"

```
temp > 30 for temp in temperatures:
```

Combining the two parts above, it's like saying, "For each temperature in the list, answer the question: Is this specific temperature greater than 30 degrees Celsius?" It creates a sequence of True or False values for each temperature.

Let's visualize how this works with a list of temperatures:

For the first temperature (25), the question is "Is 25 greater than 30?" The answer is False.

For the second temperature (28), the question is "Is 28 greater than 30?" The answer is False.

For the third temperature (30), the question is "Is 30 greater than 30?" The answer is False.

For the fourth temperature (32), the question is "Is 32 greater than 30?" The answer is True.

For the fifth temperature (29), the question is "Is 29 greater than 30?" The answer is False.

So, the result is a list [False, False, False, True, False], indicating whether each temperature is greater than 30 or not.

So the output will be:

```
At least one temperature is above 30 degrees Celsius.

Not all temperatures are above 30 degrees Celsius.
```

Next, let’s consider an example where the program checks if the entered color is present in the tuple of available colors in the colour palette and thus provides a suitable response.

let’s write a code for it:

(In Python)

```
colors_palette = ("red", "blue", "green", "yellow", "purple")

# User input: Color to search for in the palette

user_color_input = input("Enter the color you're looking for in the palette: ").lower()

# Check if the entered color is in the palette

if user_color_input in colors_palette:

print(f"Yes, the color {user_color_input} is in the color palette. You can use it for your project.")

else:
```

```
print(f"Sorry, the color {user_color_input} is not available in the color palette.")
```

Let’s suppose we enter YELLOW

the .lower() method will convert it to lowercase, and the code will correctly identify whether "yellow" is present in the colors_palette.

The in keyword in the context of if user_color_input in colors_palette is used to check if a value in this case, the value stored in the variable user_color_input is present in a sequence which in this case is the tuple colors_palette.

So the output will be:

```
Yes, the color yellow is in the color palette.
```

Let’s suppose we want to modify our output and we want to print

Yes, the color Yellow is in the color palette. Where the first letter of yellow is in upper case then we’ll just modify the print statement as

(Python)

```
if user_color_input in colors_palette:
print(f"Yes, the color {user_color_input.capitalize()} is in the color palette.")
else:
print(f"Sorry, the color {user_color_input.capitalize()} is not available in the color palette.")
```

The curly braces {} are placeholders where the value of user_color_input.capitalize() is inserted. The capitalize() method is used to ensure that the first letter of the color is in uppercase.

Moving on, just like how we use the append() method for lists, the add() method is used with sets. The add() method is used to add elements in sets in Python.

Let’s see how:

```
shopping_cart = {"apple", "banana", "orange"}
# User selects an item to add to the cart
```

```
item_to_add = input("Enter the item you want to add to your shopping cart: ").lower()

# Using the add() method to add the item to the shopping cart

shopping_cart.add(item_to_add)

# Display the updated shopping cart

print("Updated Shopping Cart:", shopping_cart)
```

In this code shopping_cart is a set with three items: "apple," "banana," and "orange" in it. The input() method then prompts the user to enter an item they want to add to the shopping cart. The lower() method is used to convert the user's input to lowercase, ensuring case-insensitive comparison.

The add() method is then used to add the user's input (item_to_add) to the shopping_cart set.

Finally, the print statement displays the updated shopping cart after the user has added an item.

Let’s run the code:

Suppose the user enters "kiwi"

The set is updated, and the output would look like this:

```
Updated Shopping Cart: {'orange', 'kiwi', 'apple', 'banana'}
```

(Run the entire code again)

Now, let's suppose the user enters "apple."

Even though "apple" is already in the set, the uniqueness property of sets ensures that duplicates are not added. The set remains the same, and the output would look like this:

```
Updated Shopping Cart: {'banana', 'apple', 'orange'}
```

Therefore, adding the same item again won't result in a duplicate entry. The set remains unchanged, and the updated shopping cart will still include the unique elements only.

Next, we have is remove() and discard(). As explained previously, the remove () method removes the first occurrence of a specified element from a list. In a similar manner remove() and discard()

method is used to remove elements, but they differ in how they handle situations where the specified element is not present in the set.

Let’s understand this with a code:

```
fruits = {"apple", "banana", "orange"}

fruits.remove("banana")

print(fruits) # Output: {'apple', 'orange'}
```

Here in this we have a set called "fruits" with three elements: "apple", "banana", and "orange".

fruits.remove("banana") removes the element "banana" from the set "fruits".

Print statement prints the updated set after removing the banana. So the output will be {'apple', 'orange'}

Now let’s suppose if you try to remove an element that is not in the set:

```
fruits = {"apple", "banana", "orange"}

fruits.remove("grape") # Raises an error

print(fruits)
```

This will raise an error because the element you're trying to remove is not found.

This can be problematic if you're not sure whether an element exists in the set before trying to remove it.

A solution to avoid the error is to use the discard method instead of remove. The discard method removes an element if it exists in the set, but it doesn't raise an error if the element is not found. Here's the modified code:

```
fruits = {"apple", "banana", "orange"}

fruits.discard("grape") # No error, even if "grape" is not in the set

print(fruits) # Output: {'apple', 'banana', 'orange'}
```

Now, if "grape" is not in the set, the discard method simply does nothing without raising an error. This can be useful when you want to remove an element but don't want the program to crash if the element is not present.

So the output of this modified code will be:

```
{'banana', 'apple', 'orange'}
```

Just like in math where you learn about unions and intersections of sets, Python's union() and intersection() work in a similar way.

Therefore the union() method returns a new set containing all the unique elements from the sets involved. It's like combining two sets while eliminating duplicate elements and intersection() method returns a new set containing only the elements that are common to all sets involved.

Let’s see it with a code:

Suppose you have two sets of fruits from two different baskets:

```
basket1 = {"apple", "banana", "orange"}

basket2 = {"banana", "grape", "kiwi"}

combined_basket = basket1.union(basket2)

print(combined_basket)
```

combined_basket set contains all the unique fruits from both basket1 and basket2. So the output will be:

```
{'apple', 'banana', 'kiwi', 'orange', 'grape'}
```

Now let’s suppose we want to find the fruits that are common between basket1 and basket2.

Using the same code in the combined basket will change union to intersection

```
basket1 = {"apple", "banana", "orange"}

basket2 = {"banana", "grape", "kiwi"}

combined_basket= basket1.intersection(basket2)

print(combined_basket)
```

Here, the only fruit that appears in both baskets is "banana," so the output will be:

```
{'banana'}
```

In Python, dictionaries have the keys() and values() methods, which can be used to retrieve the keys and values of a dictionary, respectively.

Let’s say you have a set of information about someone named Smriti. This information includes three things:

Name: Smriti

Age: 29

City: Dehradun

Now let’s imagine we have a dictionary that stores information about this person.

(In Python)

```
info={'name':'Smriti',
'age':29,
'city':'Dehradun'}
```

Now let’s suppose we want all the keys from the info dictionary. So we’ll use the keys() method which will display the keys of the dictionary, as a list.

(In Python in continuation to the above code)

```
# Retrieving keys using the keys() method
keys_list = info.keys()
# Printing the keys
print("Keys of the info dictionary:", keys_list)
```

The output will be:

```
Keys of the info dictionary: dict_keys(['name', 'age', 'city'])
```

Now, if you want to access the values corresponding to these keys, you can use the values() method:

(In Python in continuation to the above code)

```
# Retrieving values using the values() method

values_list = info.values()

# Printing the values

print("Values of the info dictionary:", values_list)
```

The output will be:

```
Values of the info dictionary: dict_values(['Smriti', 29, 'Dehradun'])
```

Alternatively, if you want both keys and values together as pairs, you can use the items() method:

(In Python)

```
# Retrieving key-value pairs using the items() method

items_list = info.items()

# Printing the key-value pairs

print("Key-Value pairs of the info dictionary:", items_list)
```

The output will be:

```
Key-Value pairs of the info dictionary: dict_items([('name', 'Smriti'), ('age', 29), ('city', 'Dehradun')])
```

These methods provide convenient ways to explore and manipulate the data stored in dictionaries in Python.

Let’s say we want to add a new key-value pair to the info dictionary. The key is 'occupation', and the corresponding value is 'Analyst'.

(In Python)

```
# Adding a new key-value pair to the dictionary

info['occupation'] = 'Analyst'

# Printing the updated dictionary

print("Updated info dictionary with occupation:", info)
```

The output will be:

```
Updated info dictionary with occupation: {'name': 'Smriti', 'age': 29, 'city': 'Dehradun', 'occupation': 'Analyst'}
```

We can also add a new key value pair using update() method as well. Let’s see how:

```
info.update({'occupation': 'Analyst'})
```

Now, let's say we want to update the age in the dictionary. We can use the update() method for this as well:

(In Python)

```
# Updating the age in the dictionary
info.update({'age': 30})
# Printing the dictionary after the update
print("Updated info dictionary with new age:", info)
```

The output will be:

```
Updated info dictionary with new age: {'name': 'Smriti', 'age': 30, 'city': 'Dehradun', 'occupation': 'Analyst'}
```

So the update() method is versatile as it can be used not only for adding new key-value pairs but also for updating existing values in the dictionary.

Let's consider another example where the update() method with another dictionary can be useful. Imagine you have a dictionary representing information about a person, and you want to update or add new details using information from another source.

Here's a scenario:

(In Python)

```
# Current information about a person
person_info = {'name': 'Alice', 'age': 25, 'city': 'New York'}
# New information from another source
additional_info = {'age': 26, 'occupation': 'Engineer', 'email': 'alice@example.com'}
```

```
# Updating the person_info dictionary with information from additional_info

person_info.update(additional_info)

# Printing the updated dictionary

print("Updated person_info dictionary:", person_info)
```

In this example:

person_info initially contains information about the person's name, age, and city.

additional_info contains new details such as an updated age, occupation, and email.

The update() method is used to incorporate the information from additional_info into person_info.

So after the update, the person_info dictionary now contains the combined information from both dictionaries.

The result is an updated dictionary that now includes the new age, occupation, and email.

The output is:

```
Updated person_info dictionary: {'name': 'Alice', 'age': 26, 'city': 'New York', 'occupation': 'Engineer', 'email': 'alice@example.com'}
```

This demonstrates how the update() method can be valuable, where you may have multiple sources of information, and you want to consolidate or update a dictionary with data from another dictionary.

In addition to updating and adding information using the update() method, there are times when you want to retrieve specific information from a dictionary. This is where the get() method comes in handy.

Let's say you want to retrieve the age of the person from the person_info dictionary:

(In Python)

```
# Using the get() method to retrieve the age

person_age = person_info.get('age')
```

```
# Printing the retrieved age
print("Retrieved age from person_info:", person_age)
```

The get() method allows you to access the value associated with a specific key in a dictionary. If the key is present, it returns the corresponding value; if the key is not present, it returns None (or a default value that you can specify). In this case, since the key 'age' is present, it will retrieve and print the age:

```
Retrieved age from person_info: 26
```

In the get() method, you can provide a default value as the second argument. If the key is present in the dictionary, it will return the corresponding value. If the key is not present, it will return the default value instead.

In this example, let's assume we are trying to retrieve the value for the key 'salary', which is not present in the person_info dictionary. By providing a default value of 'Not available', we can handle the situation gracefully:

(In Python)

```
# Using get() with a default value for a key that is not present
person_salary = person_info.get('salary', 'Not available')
# Printing the retrieved salary
print("Retrieved salary from person_info:", person_salary)
```

The output will be:

```
Retrieved salary from person_info: Not available
```

Here, even though the key 'salary' is not present in the dictionary, the get() method returns the specified default value ('Not available').

If salary was present in the dataset then the value of the salary would have been fetched and the specified default value that we have sent (‘Not available’) would not have been returned.

————

As previously discussed the remove() method eliminates the first occurrence of a specified value, for example, if you have a bag of candies, remove('chocolate') would be like taking out the first chocolate candy you find.

Now let's shift our focus to the pop() method.

With pop() method, you can either specify the position of the item to remove or, by default, it takes out the last item in the list.

In other words, when using the pop() method, you have the option to specify an index inside the parentheses. This index indicates the position of the item you want to remove and retrieve. For example, if you have a list of candies, and you use pop(2), it would remove and return the candy at the third position in the list.

However, if you don't provide any index, the pop() method defaults to removing and returning the last item in the list.

Let’s see an example to understand it better:

(In Python)

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

popped_element = lst.pop()

print(popped_element)

print(lst)
```

In the provided example, lst is a list containing the elements [1, 2, 3, 4, 5]. The pop() method is then applied to this list. The pop() method removes the last element from the list and returns it. In this case, the last element of the list is 5, so popped_element gets assigned the value 5.

So the output for the line print(popped_element) will be 5.

print(lst) # Output: [1, 2, 3, 4]: After the pop() method is applied, the original list lst is modified. The last element (5) has been removed, and the updated list is now [1, 2, 3, 4]. So print(lst) prints the modified list.

The output of the provided code would be:

```
5

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

Now, suppose we wish to remove and retrieve the element at a specified index from a list.

Let's see an example to understand how this can be achieved:

(IN PYTHON)

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

popped_element = lst.pop(2) # Removes element at index 2 (value 3)

print(popped_element)

print(lst)
```

In this example the pop() method is now called with an argument, which is the index of the element to be removed. In this case, it removes the element at index 2 from the list lst. The element at index 2 in the original list is 3, so popped_element gets assigned the value 3.

So the output of this code will be:

```
3

[1, 2, 4, 5]
```

Now, let's explore how the pop() method can be utilized with dictionaries. In a dictionary, the pop() method is employed to remove and return the value associated with a specified key. Unlike lists where we use the remove() method, dictionaries utilize pop() for this purpose. Because the remove() method is not applicable to dictionaries, the pop() method serves the purpose of removing and returning values based on specified keys in dictionary operations.

Imagine you have a dictionary representing the counts of candies in different jars, and you want to remove and retrieve the count of 'b', which is the count of chocolates that is equal to 2. Here's an example:

(In Python)

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

popped_value = my_dict.pop('b') # Removes key 'b' and returns its value (2)
```

```
print(popped_value)

print(my_dict)
```

The pop() method is applied to the dictionary my_dict with the argument 'b', which is the key to be removed. This removes the key 'b' and returns its associated value, which is 2. The removed value is then assigned to the variable popped_value.

So the output of the above code will be:

```
2

{'a': 1, 'c': 3}
```

Here the first line indicates the value associated with the removed key 'b', which is 2.

And the second line displays the modified dictionary after the removal of the key 'b', resulting in {'a': 1, 'c': 3}.

The pop() method, as discussed, removes and returns the value associated with a specified key. However, if the key is not present in the dictionary, it raises an error. To handle this situation more gracefully, you can use the pop(key, default) method.

Here is an example: using the same variable my_dictionary as above

(In Python)

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Using pop(key, default) to remove and retrieve the count of 'd' with a default value of 0

popped_value = my_dict.pop('d', 0)

print(popped_value)

print(my_dict)
```

In this example, the key 'd' is not present in the dictionary. However, by providing a default value of 0 as the second argument to pop(), we ensure that the method does not raise a Error if the key is not found. Instead, it returns the specified default value (0 in this case).

The output of the above code will be:

```
0
{'a': 1, 'b': 2, 'c': 3}
```

You can replace the default value with any alternative, such as a string like 'Key not found'. This flexibility allows you to handle missing keys in a way that best suits your program's requirements.

Let's create a simple Python program that defines a user-defined function for sending invitations. In this example, the user will enter their name, and the program will generate and print a personalized invitation message.

(In Python)

```
def send_invitation(name):
# Create an invitation message using f-string formatting
invitation_message = f"Dear {name},\n\nYou are cordially invited to our event! Please join us for an evening of fun and celebration.\n\nBest regards,\nThe Event Team"
# Print the invitation message
print(invitation_message)
# Get user input for the name
user_name = input("Enter your name: ")
# Call the send_invitation function with the user's name
send_invitation(user_name)
```

The code you provided defines a function send_invitation that takes a name as an argument, creates an invitation message using f-string formatting, and then prints the invitation message. The user is prompted to enter their name using the input function, and the entered name is then passed to the send_invitation function.

When you run this script, it will prompt you to enter your name, and then it will generate and print an invitation message using the entered name. For example let’s say the user enters Deepika. So the output will be:

```
Dear Deepika,
```

You are cordially invited to our event! Please join us for an evening of fun and celebration.

Best regards,

The Event Team

Now let’s create a dictionary representing the strength of each class in a pre-primary school.

Where the user inputs and stores the strengths of students in different classes of the school.

(In Python)

```
def class_strength_dictionary():

strength_dict = {}

for i in range(3):

class_name = input(f"Enter class name for Class {i+1}: ")

# Assuming the number of students is an integer for simplicity

num_students = int(input(f"Enter number of students for Class {i+1}: "))

strength_dict[class_name] = num_students

return strength_dict

# Create an empty dictionary to store class strengths

class_strengths = class_strength_dictionary()

# Print the resulting dictionary

print(class_strengths)
```

In this code function called class_strength_dictionary is declared. Inside this function an empty dictionary named strength_dict to store information about class strengths is initialized. A loop is started that repeats three times, with i taking on the values 0, 1, and 2. The user is then prompted to input the name of a class and the number of students in the class. strength_dict[class_name] = num_students adds an entry to the dictionary. The class name is used as the key, and the number of students is used as the value. And then at the end, the function returns the populated dictionary. class_strength_dictionary() calls the function and stores the returned dictionary in the variable class_strengths, which is then printed.

Now let’s run the code:

(Input that you’ll enter (Kaushalya))

```
Enter class name for Class 1: Nursery

Enter number of students for Class 1: 45

Enter class name for Class 2: LKG

Enter number of students for Class 2: 56

Enter class name for Class 3: UKG

Enter number of students for Class 3: 49
```

So the output will be:

```
{'Nursery': 45, 'LKG': 56, 'UKG': 49}
```

Now, let's write a program to collect names from the user, associates marks with each name, store them in a dictionary, and then displays the names and their corresponding marks.

```
def add_names_to_list():

names_list = []

for i in range(5):

name = input("Enter a name: ")

names_list.append(name)

return names_list

def create_dict_with_marks(names_list):

marks_dict = {}

for name in names_list:

marks = int(input(f"Enter marks for {name}: "))

marks_dict[name] = marks

return marks_dict
```

```
def display_name_and_marks(marks_dict):
print("Name\tMarks")
for name, marks in marks_dict.items():
print(f"{name}\t{marks}")
names_list = add_names_to_list()
marks_dict = create_dict_with_marks(names_list)
display_name_and_marks(marks_dict)
```

In this code first we have is add_names_to_list function, in which an empty list called names_list is initialized first. It then runs the loop five times, prompting the user to enter a name each time and appends the entered name to the names_list.

Finally, it returns the populated names_list.

Next we have is a create_dict_with_marks function where it takes the names_list as an argument, which is the list of names obtained from the previous function.

It then initializes an empty dictionary called marks_dict. It runs a loop through each name in the names_list and prompts the user to enter marks for that particular name.

It converts the entered marks to an integer and stores the name and marks as key-value pairs in the marks_dict. Finally, it returns the populated marks_dict.

The last is the display_name_and_marks function which takes the marks_dict as an argument, which is the dictionary containing names and corresponding marks. It first prints a header Name and Marks. It then iterates through the items in the marks_dict and prints each name and its corresponding marks in a formatted manner.

The code then calls these functions sequentially:

It first calls add_names_to_list to get a list of names.

It then calls create_dict_with_marks with the obtained names_list to create a dictionary with names and marks.

Finally, it calls display_name_and_marks to display the names and marks in a formatted way.

Now let’s run this code:

(Kaushlya: Enter values as follows:)

Enter a name: Anshul

Enter a name: Rijul

Enter a name: Swati

Enter a name: Ankit

Enter a name: Saransh

Enter marks for Anshul: 89

Enter marks for Rijul: 90

Enter marks for Swati: 76

Enter marks for Ankit: 56

Enter marks for Saransh: 65

So the output will be:

Name Marks

Anshul 89

Rijul 90

Swati 76

Ankit 56

Saransh 65

Now let's consider another scenario where function calculate_square calculates the square of a number, and function calculate_sum_of_squares uses it to calculate the sum of squares for a list of numbers. Suppose the list is=[2,4,6,8,10] so the sum of sqrs for the numbers in the list is (show it in the ppt: 2 raised to power 2 and so in) 2*2+4*2+6*2+8*2+10*2=220

Let’s see the code for it:

(In Python)

```
def calculate_square(number):
""" Calculates the square of a number."""
return number ** 2
def calculate_sum_of_squares(numbers):
""" calculates the sum of squares for a list of numbers."""
sum_of_squares = 0
for num in numbers:
square = calculate_square(num)
sum_of_squares += square
return sum_of_squares
numbers_list = [2, 4, 6, 8, 10]
result = calculate_sum_of_squares(numbers_list)
# Displaying the result
print(f"Sum of squares for {numbers_list}: {result}")
```

In this code calculate_square function takes a number as input and calculates its square using the ** operator.

calculate_sum_of_squares function takes a list of numbers as input, iterates through each number, and calls calculate_square to calculate the square. It then accumulates these squares to compute the sum of squares.

A list of numbers called numbers_list is created with values 2,4,6,8,10 and calculate_sum_of_squares function is then called with this list.

The result is then printed, showing the sum of squares for the given list of numbers.

So the output of the code will be:

```
220
```

Now let’s write a code to square the numbers in the list. But first let’s understand about map() function.

map() function is like a magic wand for an iterable. It helps you perform the same action on every item in an iterable without needing to write a loop.

Imagine you have a list of numbers, and you want to do something to each number, like double them or square them. Instead of going through the list one by one, you can use the map() function.

Here's the basic syntax of the map() function:

```
map(function, iterable)
```

Now let’s proceed by writing a code to square the numbers in the list using map() function.

(In Python)

```
def square_number(x):
return x ** 2
# List of numbers
numbers = [1, 2, 3, 4, 5]
# Using map() to apply the square_number function to each number in the list
squared_numbers = list(map(square_number, numbers))
# Displaying the original and squared numbers
print("Original numbers:", numbers)
print("Squared numbers:", squared_numbers)
```

In this code the square_number function is defined to square a given number. The map() function is then used which takes two arguments: the function (square_number) and the iterable (numbers). This means we want to apply the square_number function to each number in the numbers list. The result is a new list named squared_numbers.

Finally, the original list of numbers and the list of squared numbers are printed to the console.

So the output for the above code will be:

```
Original numbers: [1, 2, 3, 4, 5]

Squared numbers: [1, 4, 9, 16, 25]
```

Now let’s write this code in a concise manner using lambda function as discussed previously:

(In Python)

```
numbers = [1, 2, 3, 4, 5]

# Using lambda function to square each number in the list

squared_numbers_lambda = list(map(lambda x: x ** 2, numbers))

# Displaying the original and squared numbers using lambda function

print("Original numbers:", numbers)

print("Squared numbers:", squared_numbers_lambda)
```

In this code the lambda function takes an input x and returns the square of x (denoted as x ** 2). The map() function applies the lambda function to each element of the iterable (numbers).

For each number in the list, it calculates the square using the lambda function. The list() function is used to convert the map object into a list of squared numbers that can be assigned to squared_numbers_lambda.

So the output of the above code will be:

```
Original numbers: [1, 2, 3, 4, 5]

Squared numbers : [1, 4, 9, 16, 25]
```