Q1.**What is the difference between a function and a method in Python?**

ANS. Python, methods and functions have similar purposes but differ in important ways. Functions are independent blocks of code that can be called from anywhere, while methods are tied to objects or classes and need an object or class instance to be invoked. Here, key differences between Method and Function in Python are explained. Java is also an OOP language, but there is no concept of Function in it. But Python has both concept of Method and Function. Python Method Method is called by its name, but it is associated to an object (dependent). A method definition always includes 'self' as its first parameter. A method is implicitly passed to the object on which it is invoked. It may or may not return any data. A method can operate on the data (instance variables) that is contained by the corresponding class

Basic Method Structure in Python :

```
# Basic Python method
class class_name
    def method_name () :
        ......
        # method body
        ......
```

```
  File "<ipython-input-2-4bb9c699dc14>", line 2
    class class_name
                    ^
SyntaxError: expected ':'
```

User-Defined Method :

```
# Python 3  User-Defined  Method
class ABC :
    def method_abc (self):
        print("I am in method_abc of ABC class. ")

class_ref = ABC() # object of ABC class
class_ref.method_abc()
```

```
I am in method_abc of ABC class.
```

Inbuilt method :

```
import math

ceil_val = math.ceil(15.25)
print( "Ceiling value of 15.25 is : ", ceil_val)
```

```
Ceiling value of 15.25 is :  16
```

```
          **Functions**
```

Function is block of code that is also called by its name. (independent) The function can have different parameters or may not have any at all. If any data (parameters) are passed, they are passed explicitly. It may or may not return any data. Function does not deal with Class and its instance concept.

Basic Function Structure in Python :

```
def function_name ( arg1, arg2, ...) :
    ......
    # function body
    ......
```

User-Defined Function :

```
def Subtract (a, b):
    return (a-b)
```

```
print( Subtract(10, 12) ) # prints -2

print( Subtract(15, 6) ) # prints 9
```

```
-2
9
```

Inbuilt Function :

```
s = sum([5, 15, 2])
print( s ) # prints 22

mx = max(15, 6)
print( mx ) # prints 15
```

```
22
15
```

Difference between method and function

Simply, function and method both look similar as they perform in almost similar way, but the key difference is the concept of ‘Class and its Object‘. Functions can be called only by its name, as it is defined independently. But methods can’t be called by its name only, we need to invoke the class by a reference of that class in which it is defined, i.e. method is defined within a class and hence they are dependent on that class.

Q2. **Explain the concept of function arguments and parameters in Python?**

ANS. Function arguments are values passed to a function when it is called. On the other hand, parameters are the variables inside the function's parentheses. There are four types of function arguments in Python: required arguments, default arguments, keyword arguments, and arbitrary arguments.Arguments Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma. The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

Arguments are often shortened to args in Python documentations.

Parameters or Arguments?

The terms parameter and argument can be used for the same thing: information that are passed into a function.

From a function's perspective: A parameter is the variable listed inside the parentheses in the function definition.An argument is the value that are sent to the function when it is called. Number of Arguments By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less. **Example** This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

```
Emil Refsnes
```

If you try to call the function with 1 or 3 arguments, you will get an error: **Example** This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-49655f8043be> in <cell line: 4>()
      2   print(fname + " " + lname)
      3
----> 4 my_function("Emil")

TypeError: my_function() missing 1 required positional argument: 'lname'
```

Q3.What are the different ways to define and call a function in Python?

ANS. There are a few ways to define a function in Python. Using the def keyword followed by the function name, parenthesis for arguments and a colon. Add the function body with an indentation. You can optionally use the return keyword if you want the function to return a value.

```
def my_function(arg1, arg2):
  # function body
  return arg1 + arg2
```

To call a function, simply type the function name followed by parenthesis and any arguments to be passed. my_function(5, 6) In programming, a function is a reusable block of code that executes a certain functionality when it is calleDd.Functions are integral parts of every programming language because they help make your code more modular and reusable. In this article, I will show you how to define a function in Python and call it, so you can break down the code of your Python applications into smaller chunks.I will also show you how arguments and the return keyword works in Python functions.

Basic Syntax for Defining a Function in Python

In Python, you define a function with the def keyword, then write the function identifier (name) followed by parentheses and a colon. he next thing you have to do is make sure you indent with a tab or 4 spaces, and then specify what you want the function to do for you.

```
  def functionName():
# What to make the function do
```

Basic Examples of a Function in Python

Following the basic syntax above, an example of a basic Python function printing “Hello World” to the terminal looks like this:

```
def myfunction():
print("Hello World")
```

To call this function, write the name of the function followed by parentheses:

```
myfunction()
```

Next, run your code in the terminal by typing python filename.py to show what you want the function to do: Another basic example of subtractig 2 numbers looks like this:

```
def subtractNum():
print(34 - 4)
subtractNum()
# Output: 30
```

Arguments in Python Functions While defining a function in Python, you can pass argument(s) into the function by putting them inside the parenthesis.

The basic syntax for doing this looks as shown below: def functionName(arg1, arg2):

```
# What to do with function
```

When the function is called, then you need to specify a value for the arguments:

```
functionName(valueForArg1, valueForArg2)
```

Here's an example of arguments in a Python function:

```
def addNum(num1, num2):
print(num1 + num2)
addNum(2, 4)
# Output: 6
```

In the example above:

I passed 2 arguments into the function named addNum I told it to print the sum of the 2 arguments to the terminal I then called it with the values for the 2 arguments specified N.B.: You can specify as many arguments as you want.

How to Use the Return Keyword in Python In Python, you can use the return keyword to exit a function so it goes back to where it was called. That is, send something out of the function.

The return statement can contain an expression to execute once the function is called.The example below demonstrates how the return keyword works in Python:

```
def multiplyNum(num1):
return num1 * 8
result = multiplyNum(8)
print(result)
```

Output: 64

What’s the code above doing?

I defined a function named multiplyNum and passed it num1 as an argument Inside the function, I used the return keyword to specify that I want num1 to be multiplied by 8 After that, I called the function, passed 8 into it as the value for theAfter that, I called the function, passed 8 into it as the value for the num1 argument, and assigned the function call to a variable I named result With the result variable, I was able to print what I intended to do with the function to the terminal

Conclusion

In this article, you learned how to define and call functions in Python. You also learned how to pass arguments into a function and use the return keyword, so you can be more creative with the functions you write. If you find this article helpful, don’t hesitate to share it with your friends and family.

Q4.. **What is the purpose of the `return` statement in a Python function?**

ANS.The return statement in Python serves two primary purposes:

Ending Function Execution: When a return statement is encountered within a function, it immediately terminates the function's execution. The code following the return statement within the function will not be executed.

Returning a Value: return is used to send a value back from the function to the caller. This value can be any data type, or even another function. If no value is specified after return or if there's no return statement at all, the function implicitly returns None.

```
def add(x, y):
  sum = x + y
  return sum

result = add(5, 3)
print(result)
```

```
8
```

A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned. A return statement is overall used to invoke a function so that the passed statements can be executed.

Note: Return statement can not be used outside the function.

```
  Syntax:
 def fun():
 statements
 .
 .
 return [expression]
```

Example: def cube(x): r=x**3 return r

```
# Python program to
# demonstrate return statement

def add(a, b):

    # returning sum of a and b
    return a + b

def is_true(a):

    # returning boolean of a
    return bool(a)

# calling function
res = add(2, 3)
print("Result of add function is {}".format(res))

res = is_true(2<5)
print("\nResult of is_true function is {}".format(res))
```

```
Result of add function is 5

Result of is_true function is True
```

Returning Multiple Values In Python, we can return multiple values from a function. Following are different ways.Using Object: This is similar to C/C++ and Java, we can create a class (in C, struct) to hold multiple values and return an object of the class.

```
# A Python program to return multiple
# values from a method using class
class Test:
    def __init__(self):
        self.str = "geeksforgeeks"
        self.x = 20

# This function returns an object of Test
def fun():
    return Test()

# Driver code to test above method
t = fun()
print(t.str)
print(t.x)
```

```
geeksforgeeks
20
```

Using a list: A list is like an array of items created using square brackets. They are different from arrays as they can contain items of different types. Lists are different from tuples as they are mutable. See this for details of list.

```
# A Python program to return multiple
# values from a method using list

# This function returns a list
def fun():
    str = "geeksforgeeks"
    x = 20
    return [str, x];

# Driver code to test above method
```

```
list = fun()
print(list)
```

```
['geeksforgeeks', 20]
```

Q5.**What are iterators in Python and how do they differ from iterables?**

ANS. In Python, iterables and iterators are essential concepts for working with sequences of data. They provide a powerful way to traverse and process elements in collections like lists, tuples, strings, and more. While closely related, iterables and iterators have distinct roles and functionalities.

Iterables

An iterable is any Python object capable of returning its members one at a time, allowing it to be iterated over in a for loop. In simpler terms, an iterable is something you can loop over. Common examples of iterables include lists, tuples, strings, dictionaries, files, and generators. These objects have an inbuilt method called **iter**() that returns an iterator object.

Iterators

An iterator is an object that represents a stream of data. It returns one data element at a time. It also remembers its position during iteration. Unlike iterables, iterators don't have all the elements readily available in memory. They generate the elements on demand as you iterate through them.

Iterators use the **next**() method to retrieve the next item in the sequence. When there are no more items, it raises a StopIteration exception, signaling the end of the iteration.

```
**Key Differences**
```

Definition:

An iterable is an object with an **iter**() method that returns an iterator. An iterator is an object with a **next**() method that returns the next value in the sequence.

Functionality:

Iterables are objects that can be iterated over. Iterators are objects that facilitate the iteration process.

Memory:

Iterables may store all their elements in memory. Iterators generate elements on demand, saving memory.

Methods:

Iterables have the **iter**() method. Iterators have the **iter**() and **next**() methods.

```
# Create an iterable (list)
my_list = [1, 2, 3, 4, 5]

# Get an iterator from the iterable
my_iterator = iter(my_list)

# Iterate through the elements using the iterator
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
```

```
1
2
3
```

In this example, my_list is an iterable. We obtain an iterator (my_iterator) from it using the iter() function. Then, we use the next() function to retrieve each element from the iterator.

```
**Benefits of Iterators**
```

Lazy Evaluation: Iterators generate values only when needed, saving memory and improving efficiency, especially for large datasets.

State Maintenance: Iterators remember their position, allowing you to resume iteration from where you left off.

Composability: Iterators can be chained and combined to create complex data processing pipelines.

Iterators and iterables are fundamental concepts in Python, enabling efficient and flexible data processing. Understanding their differences and how to use them effectively is crucial for writing clean and optimized code.

Q6. **Explain the concept of generators in Python and how they are defined.**

ANS. In Python, generators are a powerful and efficient way to create iterators. They provide a concise syntax for defining iterators without the need to write a separate iterator class with **iter**() and **next**() methods. Generators are especially useful for working with large datasets or infinite sequences where storing all the elements in memory would be impractical.

Concept of Generators

Generators are functions that use the yield keyword instead of return. When a generator function is called, it doesn't execute the entire function body at once. Instead, it returns a generator object, which is an iterator. Each time the next() method is called on the generator object, the function resumes execution from where it left off, yielding the next value in the sequence. This process continues until the generator function either returns or raises a StopIteration exception.

Defining Generators

Defining a generator function is similar to defining a regular function, but with the key difference of using the yield keyword. The yield keyword suspends the function's execution and returns a value to the caller. When the generator is resumed, it continues execution from the point where it yielded.

Here's a simple example of a generator function that generates a sequence of numbers:

```
def number_generator(n):
  for i in range(n):
    yield i

# Create a generator object
gen = number_generator(5)

# Iterate over the generator
for num in gen:
  print(num)
```

```
0
1
2
3
4
```

n this example, the number_generator function is a generator. When called with the argument 5, it returns a generator object gen. The for loop iterates over the generator, and each time the next() method is called implicitly, the generator function resumes, yields the next number in the sequence, and suspends again.

Benefits of Generators

Generators offer several advantages over traditional iterators:

Lazy Evaluation:

Generators generate values only when needed, saving memory and improving efficiency, especially for large datasets. They avoid creating and storing the entire sequence in memory beforehand.

**Improved Readability: *

Generators often lead to more concise and readable code compared to writing separate iterator classes. The use of yield makes the code flow more intuitive.

Infinite Sequences:

Generators can represent infinite sequences without running into memory issues. They generate values on demand, allowing you to work with sequences that would be impossible to store entirely.

Pipeline Processing:

Generators can be chained together to create complex data processing pipelines. The output of one generator can be fed as input to another, allowing for modular and efficient data manipulation.

Common Use Cases

Generators are widely used in various scenarios, including:

Data Processing: Generators are ideal for processing large datasets, such as reading files line by line or filtering data streams.

Infinite Sequences: Generators can represent mathematical sequences, such as Fibonacci numbers or prime numbers, without having to pre-compute and store the entire sequence.

**Coroutines: * Generators can be used to implement coroutines, which are functions that can be paused and resumed, allowing for concurrent execution.

**Custom Iterators: * Generators provide a convenient way to create custom iterators for specific data structures or algorithms.

Conclusion

Generators are a powerful tool in Python for creating iterators. They offer lazy evaluation, improved readability, and the ability to represent infinite sequences. By understanding the concept of generators and how to define them, you can write more efficient and elegant code for various tasks involving data processing and iteration.

Q7. **What are the advantages of using generators over regular functions?**

In Python, both generators and regular functions are essential tools for defining reusable blocks of code, but they differ significantly in how they handle data and execution flow. Generators offer several advantages over regular functions, particularly when dealing with large datasets, infinite sequences, or situations where lazy evaluation is desired.

1. Lazy Evaluation: Memory Efficiency and Performance

One of the primary advantages of generators is their ability to perform lazy evaluation. Unlike regular functions that compute and return all their results at once, generators produce values on demand, one at a time, only when requested. This has significant implications for memory efficiency and performance, especially when dealing with large datasets or complex computations.

Reduced Memory Consumption: Regular functions often require storing the entire result set in memory before returning it. This can be problematic for large datasets, leading to excessive memory usage and potential performance bottlenecks. Generators, on the other hand, generate values only when needed, avoiding the need to store the complete result set in memory. This makes them highly memory-efficient, particularly for tasks involving large or infinite sequences.

Improved Performance: In scenarios where not all results are required, generators can significantly improve performance. By generating values on demand, they avoid unnecessary computations and memory allocations, leading to faster execution times. For instance, if you only need the first few elements of a large sequence, a generator can provide them without having to compute the entire sequence beforehand.

2. Enhanced Readability and Code Conciseness

Generators often lead to more concise and readable code compared to regular functions, especially when dealing with iterative processes. The use of the yield keyword simplifies the logic of generating values, making the code flow more intuitive and easier to understand.

Simplified Iteration: Generators inherently support iteration. They can be used directly in for loops or with other iterative constructs without the need for explicit indexing or manual iteration management. This reduces code complexity and enhances readability, making the code more expressive and easier to maintain.

Concise Syntax: The yield keyword provides a concise way to define generators, eliminating the need for writing separate iterator classes with **iter**() and **next**() methods. This reduces code verbosity and makes generator definitions more compact and easier to grasp.

3. Representation of Infinite Sequences

Generators excel in representing infinite sequences, something that is not feasible with regular functions. Regular functions have a finite execution path and eventually return a result. Generators, with their ability to suspend and resume execution, can generate values indefinitely without running into memory issues.

Mathematical Sequences: Generators are well-suited for representing mathematical sequences like Fibonacci numbers or prime numbers. They can generate these sequences on demand, allowing you to work with them without having to pre-compute and store the entire sequence.

Data Streams: Generators can model data streams, such as continuous sensor readings or network packets. They can generate values as they become available, providing a flexible way to handle real-time data processing.

4. Pipeline Processing and Data Manipulation

Generators facilitate pipeline processing, allowing you to chain multiple generators together to create complex data manipulation workflows. The output of one generator can be fed as input to another, enabling modular and efficient data transformations.

Data Filtering and Transformation: Generators can be used to filter, transform, and process data in a step-by-step manner. Each generator in the pipeline can perform a specific operation on the data, creating a modular and reusable data processing pipeline.

Data Aggregation and Analysis: Generators can be combined with other functions and libraries to perform data aggregation and analysis. For example, you can use generators to process data streams and feed the results into analytical functions or machine learning models.

5. Coroutines and Concurrent Execution

Generators can be used to implement coroutines, which are functions that can be paused and resumed, allowing for concurrent execution. This is particularly useful for tasks involving asynchronous operations or event handling.

Asynchronous Programming: Generators can be used to write asynchronous code, where operations can be performed concurrently without blocking the main execution thread. This is essential for applications that need to respond to events or handle multiple tasks simultaneously.

Event Handling: Generators can be used to create event handlers that respond to specific events or triggers. The generator can suspend execution until an event occurs and then resume to process the event data.

Conclusion

Generators offer several compelling advantages over regular functions, making them a valuable tool in Python for various tasks. Their lazy evaluation, concise syntax, ability to represent infinite sequences, and support for pipeline processing make them ideal for memory-efficient data processing, complex iterations, and concurrent execution scenarios. By understanding the benefits of generators, you can write more efficient, readable, and flexible code for a wide range of applications.

Q8.**What is a lambda function in Python and when is it typically used?**

Lambda functions in Python are anonymous, small, and single-expression functions defined using the lambda keyword. They are also known as anonymous functions because they don't have a formal name like regular functions defined with the def keyword.

Lambda functions are typically used when you need a simple function for a short period and don't want to define a full function using the def keyword. They are often used as arguments to higher-order functions like map, filter, and reduce.

Syntax of Lambda Functions

The syntax of a lambda function is as follows:

lambda arguments: expression

lambda: The keyword used to define a lambda function. arguments: One or more arguments passed to the function, similar to regular functions. expression: A single expression that is evaluated and returned by the function.

Example

```
add = lambda x, y: x + y
result = add(5, 3)
print(result)
```

```
8
```

Key Characteristics of Lambda Functions

Anonymous: Lambda functions don't have a formal name like regular functions defined with def. They are often used directly within expressions or as arguments to other functions.

Single Expression: Lambda functions are limited to a single expression, which is evaluated and returned. This makes them suitable for simple operations or calculations.

Concise: Lambda functions provide a compact way to define functions, reducing code verbosity and making code more readable in certain situations.

Short-Lived: Lambda functions are typically used for short-lived operations or as temporary functions within a larger context.

Typical Use Cases of Lambda Functions

Lambda functions are commonly used in the following scenarios:

Higher-Order Functions: Lambda functions are frequently used as arguments to higher-order functions, which are functions that take one or more functions as arguments or return a function. Common higher-order functions in Python include map, filter, and reduce.

map: Applies a function to each item in an iterable (e.g., a list) and returns a new iterable with the results.

Q9. Explain the purpose and usage of the `map()` function in Python.

ANS In Python, the map() function is a built-in higher-order function that provides a powerful and efficient way to apply a given function to each item in an iterable (e.g., a list, tuple, or string) and return a new iterable containing the transformed values. It offers a concise and functional approach to data processing, eliminating the need for explicit loops and enhancing code readability.

The primary purpose of the map() function is to apply a function to each element of an iterable without explicitly writing a loop. This functional programming paradigm promotes code reusability, reduces code verbosity, and often leads to more efficient execution, especially for large datasets.

The basic syntax of the map() function is as follows: map(function, iterable, ...)

function: The function to be applied to each element of the iterable. This can be a built-in function, a *user*-defined function, or a lambda function. iterable: The iterable containing the elements to be processed. This can be a list, tuple, string, or any other iterable object. *...: *Optional additional iterables. If more than one iterable is provided, the function will be applied to the corresponding elements from all iterables. The map() function returns a map object, which is an iterator. To access the transformed values, you can iterate over the map object using a loop or convert it to a list, tuple, or another iterable type.

Here's a breakdown of the usage of the map() function with illustrative examples:

1. Applying a Function to a Single Iterable

The most common use case of map() is to apply a function to each element of a single iterable. For instance, let's say you have a list of numbers and want to square each number:

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

def square(x):
  return x * x

squared_numbers = map(square, numbers)

print(list(squared_numbers))
```

```
[1, 4, 9, 16, 25]
```

In this example, the square() function is applied to each number in the numbers list using map(). The resulting map object is then converted to a list to display the squared numbers.

2. Using Lambda Functions with map()

Lambda functions, being anonymous and concise, are often used with map() to define simple operations without the need for separate function definitions. For example, to convert a list of strings to uppercase:

```
strings = ["hello", "world"]

uppercase_strings = map(lambda s: s.upper(), strings)

print(list(uppercase_strings))
```

```
['HELLO', 'WORLD']
```

3. Applying a Function to Multiple Iterables

The map() function can also be used to apply a function to corresponding elements from multiple iterables. For instance, to add corresponding elements from two lists:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

sum_lists = map(lambda x, y: x + y, list1, list2)

print(list(sum_lists))
```

```
[5, 7, 9]
```

4. Benefits of Using map()

The map() function offers several benefits:

Concise and Readable Code: map() eliminates the need for explicit loops, making the code more concise and easier to read. Functional Programming Paradigm: map() promotes a functional programming style, where functions are treated as first-class citizens and can be passed as arguments to other functions. Potential for Optimization: map() can be optimized internally for better performance, especially for large datasets. Lazy Evaluation: map() returns an iterator, which performs lazy evaluation, generating values only when needed. This can be beneficial for memory efficiency when dealing with large datasets.

5. Common Use Cases

The map() function is widely used in various scenarios:

Data Transformation: Applying a function to transform elements in a list, such as converting data types, cleaning data, or performing calculations. Parallel Processing: map() can be used with libraries like multiprocessing to apply a function to elements in parallel, potentially speeding up execution for large datasets. Functional Programming: map() is a fundamental tool in functional programming paradigms, enabling concise

10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

ANS In Python, map(), reduce(), and filter() are built-in higher-order functions that operate on iterables (like lists, tuples, etc.) and provide a functional approach to data processing. They offer a concise and efficient way to perform common operations on sequences of data, eliminating the need for explicit loops and enhancing code readability. While they all work with iterables, they have distinct purposes and functionalities.

1. map()

Purpose: The map() function applies a given function to each item in an iterable and returns a new iterable containing the transformed values. It's like taking a blueprint (the function) and applying it to each item in a collection to create a new, modified collection.

Syntax: map(function, iterable, ...)

- `function`: The function to be applied to each element.
- `function`: The function to be applied to each element.
- `iterable`: The iterable containing the elements to be processed.
- `...`: Optional additional iterables (the function will be applied to corresponding elements from all iterables).

EXAMPLE

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

def square(x):
  return x * x

squared_numbers = map(square, numbers)

print(list(squared_numbers))
```

```
[1, 4, 9, 16, 25]
```

In this example, the square() function is applied to each number in the numbers list using map(). The resulting map object is then converted to a list to display the squared numbers.

2. reduce()

Purpose: The reduce() function applies a given function cumulatively to the items of an iterable, reducing them to a single value. It's like taking a collection of items and repeatedly combining them using a specific operation until you have a single result.

Syntax: reduce(function, iterable[, initializer])

function: The function to be applied cumulatively to the items. It takes two arguments: the accumulated value so far and the next item in the iterable. iterable: The iterable containing the elements to be processed.

EXAMPLE

```
from functools import reduce

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

```
def add(x, y):
  return x + y

sum_of_numbers = reduce(add, numbers)

print(sum_of_numbers)
```

```
15
```

In this example, the add() function is applied cumulatively to the numbers in the numbers list using reduce(). The result is the sum of all the numbers in the list.

3. filter()

Purpose: The filter() function constructs an iterator from those elements of an iterable for which a given function returns True. It's like using a sieve to select only the items from a collection that meet a specific condition.

Syntax: filter(function, iterable)

function: The function to be applied to each element. It should return True if the element should be included in the result, and False otherwise.

EXAMPLE

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

def is_even(x):
  return x % 2 == 0

even_numbers = filter(is_even, numbers)

print(list(even_numbers))
```

```
[2, 4]
```

In this example, the is_even() function is used to filter the numbers list, selecting only the even numbers. The resulting filter object is then converted to a list to display the even numbers.

11. Using pen & Paper write the internal mechanism for sum operation using reduce function on this given ## list:[47,11,42,13];

```
from google.colab import files
uploaded = files.upload()
print(uploaded)
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving WhatsApp Image 2024-09-12 at 10.30.43 AM.jpeg to WhatsApp Image 2024-09-12 at 10.30.43 AM (1).jpeg
{'WhatsApp Image 2024-09-12 at 10.30.43 AM (1).jpeg': b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x00\x00\x01\x00\x01\x00\x00\xff\xdb\x0
```

Practical Questions:

1. Write a Python function that takes a list of numbers as input and returns the sum of all even numbers in the list.

```
def sum_even_numbers(numbers):
  """
  This function takes a list of numbers as input and returns the sum of all even numbers in the list.

  Args:
    numbers: A list of numbers.
  Returns:
    The sum of all even numbers in the list.
  """

  sum = 0
  for number in numbers:
    if number % 2 == 0:
      sum += number
  return sum
  # Example usage
numbers = [1, 2, 3, 4, 5, 6]
even_sum = sum_even_numbers(numbers)
print(f"The sum of even numbers in the list is: {even_sum}")
```

```
The sum of even numbers in the list is: 12
```

This function, named sum_even_numbers, takes a list of numbers as input and calculates the sum of all even numbers present in the list. It initializes a variable sum to 0, which will store the cumulative sum of even numbers. The function then iterates through each number in the input list using a for loop. For each number, it checks if the number is even using the modulo operator (%). If the number is even (i.e., the remainder when divided by 2 is 0), it is added to the sum variable. Finally, the function returns the sum of all even numbers.

This function provides a clear and concise solution to the problem of calculating the sum of even numbers in a list. It demonstrates the use of a for loop to iterate through the list, an if statement to check for even numbers, and the modulo operator to determine divisibility by 2. The function is well-defined with a descriptive name, a docstring explaining its purpose and usage, and an example demonstrating how to call the function and print the result.

The example usage provided shows how to create a list of numbers, call the sum_even_numbers function with the list as an argument, and print the returned sum of even numbers. This example helps to illustrate how the function can be used in practice and provides a clear understanding of its functionality.

This function can be easily modified or extended to handle different scenarios or requirements. For instance, you could modify the function to calculate the sum of odd numbers, or you could add a parameter to specify the divisor for checking divisibility. The function can also be used as a building block for more complex functions or algorithms that involve processing lists of numbers

2. Create a Python function that accepts a string and returns the reverse of that string.

```
def reverse_string(s):
  return s[::-1]

# Example usage
string = "Hello, World!"
reversed_string = reverse_string(string)
print(f"The reversed string is: {reversed_string}")
```

```
The reversed string is: !dlroW ,olleH
```

3. Implement a Python function that takes a list of integers and returns a new list containing the squares of each number?

```
def square_numbers(numbers):
  """
  This function takes a list of integers and returns a new list containing the squares of each number.

  Args:
    numbers: A list of integers.

  Returns:
    A new list containing the squares of each number in the input list.
  """

  squared_numbers = []
  for number in numbers:
    squared_numbers.append(number * number)
  return squared_numbers
```

```
# Example usage
numbers = [1, 2, 3, 4, 5]
squared_numbers = square_numbers(numbers)
print(f"The squared numbers are: {squared_numbers}")
```

```
The squared numbers are: [1, 4, 9, 16, 25]
```

4. Write a Python function that checks if a given number is prime or not from 1 to 200.

```
def is_prime(n):
  """
  This function checks if a given number is prime or not.

  Args:
    n: The number to be checked.

  Returns:
    True if the number is prime, False otherwise.
  """
  if n <= 1:
    return False
  if n <= 3:
    return True
  if n % 2 == 0 or n % 3 == 0:
    return False
  i = 5
  while i * i <= n:
    if n % i == 0 or n % (i + 2) == 0:
      return False
    i = i + 6
  return True

# Test the function
for num in range(1, 201):
  if is_prime(num):
    print(num)
```

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
101
103
107
109
113
127
131
137
139
149
151
157
163
167
173
179
```

```
181
191
193
197
199
```

5. Create an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms.

```
class FibonacciIterator:
    def __init__(self, n):
        self.n = n
        self.a = 0
        self.b = 1
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count < self.n:
            result = self.a
            self.a, self.b = self.b, self.a + self.b
            self.count += 1
            return result
        else:
            raise StopIteration

# Create a Fibonacci iterator for 10 terms
fib_iter = FibonacciIterator(10)

# Iterate over the iterator and print the values
for num in fib_iter:
    print(num)
```

```
0
1
1
2
3
5
8
13
21
34
```

6. Write a generator function in Python that yields the powers of 2 up to a given exponent.

```
def powers_of_2(exponent):
  for i in range(exponent + 1):
    yield 2 ** i

# Example usage
exponent = 10
for power in powers_of_2(exponent):
  print(power)
```

```
1
2
4
8
16
32
64
128
256
512
1024
```

7. Implement a generator function that reads a file line by line and yields each line as a string.

```
# A generator function that yields 1 for first time,
# 2 second time and 3 third time
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

# Driver code to check above generator function
for value in simpleGeneratorFun():
    print(value)
```

```
1
2
3
```

8. Use a lambda function in Python to sort a list of tuples based on the second element of each tuple.

```
tuples = [(1, 5), (3, 2), (2, 8)]
sorted_tuples = sorted(tuples, key=lambda x: x[1])
print(sorted_tuples)
```

```
[(3, 2), (1, 5), (2, 8)]
```

9. Write a Python program that uses `map()` to convert a list of temperatures from Celsius to Fahrenheit.

```
# Define the list of temperatures in Celsius
celsius_temperatures = [0, 10, 20, 30, 40]

# Define a function to convert Celsius to Fahrenheit
def celsius_to_fahrenheit(celsius):
  return (celsius * 9/5) + 32

# Use map() to apply the conversion function to each temperature
fahrenheit_temperatures = map(celsius_to_fahrenheit, celsius_temperatures)

# Convert the map object to a list to see the results
print(list(fahrenheit_temperatures))
```

```
[32.0, 50.0, 68.0, 86.0, 104.0]
```

10. Create a Python program that uses `filter()` to remove all the vowels from a given string.

```
def remove_vowels(text):
  vowels = "aeiouAEIOU"
  return ''.join(filter(lambda char: char not in vowels, text))

# Example usage
my_string = "Hello, World!"
result = remove_vowels(my_string)
print(result)
```

```
Hll, Wrld!
```

11. Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:

Write a Python program, which returns a list with 2-tuples. Each tuple consists of the order number and the product of the price per item and the quantity. The product should be increased by 10,- € if the value of the order is smaller than 100,00 €.

Write a Python program using lambda and map.

```
orders = [ ["34587", "Learning Python, Mark Lutz", 4, 40.95],
           ["98762", "Programming Python, Mark Lutz", 5, 56.80],
           ["77226", "Head First Python, Paul Barry", 3,32.95],
           ["88112", "Einführung in Python3, Bernd Klein",  3, 24.99]]

min_order = 100
invoice_totals = list(map(lambda x: x if x[1] >= min_order else (x[0], x[1] + 10),
                          map(lambda x: (x[0],x[2] * x[3]), orders)))
print(invoice_totals)
```

```
[('34587', 163.8), ('98762', 284.0), ('77226', 108.85000000000001), ('88112', 84.97)]
```