Theory Questions

Que_1. What is the difference between a function and a method in Python?

Functions vs. Methods in Python

While both functions and methods are used to encapsulate reusable code in Python, there's a key distinction between them:

Functions:

- Are standalone entities that can be defined outside of classes.
- Can be called independently without being associated with any object.
- Used for general-purpose code that can be reused in different contexts.

Example:

```
def greet(name):
    print("Hello, " + name + "!")

greet("Alice")  # Output: Hello, Alice!
```

Methods:

- Are functions that are defined within a class.
- Are associated with specific objects and can access and modify the object's attributes.
- Used to implement the behavior of objects.

Example:

```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print("Hello, " + self.name + "!")

person = Person("Bob")
person.greet()  # Output: Hello, Bob!
```

In summary:

- **Functions** are standalone entities that can be called independently.
- **Methods** are functions that belong to a class and are associated with objects.

When to use functions or methods:

- Use **functions** for general-purpose code that can be reused in different contexts.

- Use **methods** to define the behavior of objects and encapsulate related functionality within a class.

Que_2. Explain the concept of function arguments and parameters in Python.

Function Arguments and Parameters in Python

In Python, functions are used to encapsulate reusable code blocks. They can take input values, known as **arguments**, and return output values.

Parameters:

- **Formal parameters:** These are the variables defined within the function's parentheses. They act as placeholders for the actual values that will be passed to the function.
- **Default parameters:** Parameters can have default values assigned to them. If a value is not provided for a parameter when the function is called, the default value is used.

Arguments:

- **Actual arguments:** These are the values that are passed to the function when it is called. They are assigned to the corresponding parameters.
- **Keyword arguments:** Arguments can be passed using keyword arguments, where the parameter name is explicitly specified followed by the value. This allows the order of arguments to be flexible.

Example:

```
def greet(name, greeting="Hello"):
    print(greeting, name)

# Calling the function with positional arguments
greet("Alice")  # Output: Hello Alice

# Calling the function with keyword arguments
greet(greeting="Hi", name="Bob")  # Output: Hi Bob
```

In this example:

- `name` and `greeting` are the parameters defined in the function.
- `"Alice"` and `"Bob"` are the arguments passed to the function when it is called.
- The `greeting` parameter has a default value of `"Hello"`, so it can be omitted when calling the function.

Key Points:

- The number of arguments passed to a function must match the number of parameters defined in the function, unless default values are provided.
- Arguments can be passed by position or by keyword.
- Keyword arguments allow for more flexibility in the order of arguments.
- Functions can return values using the `return` statement.

Understanding function arguments and parameters is essential for writing modular and reusable code in Python.

Que_3. What are the different ways to define and call a function in Python?

Defining Functions in Python

There are two primary ways to define functions in Python:

1. **Using the `def` keyword:**

   ```
   def function_name(parameters):
       # Function body
       # Statements to be executed
       return value  # Optional return statement
   ```

   - `function_name`: The name you give to the function.
   - `parameters`: Optional parameters that the function can accept.
   - `return value`: Optional value that the function returns.
2. **Using lambda expressions (anonymous functions):**

   ```
   lambda parameters: expression
   ```

   - A concise way to define a function without giving it a name.

Calling Functions

To call a function, you simply use its name followed by parentheses containing the arguments (if any).

```
function_name(arguments)
```

Examples:

```
# Defining a function using `def`
def greet(name):
    print("Hello, " + name + "!")

# Calling the function
greet("Alice")

# Defining a lambda function
add = lambda x, y: x + y

# Calling the lambda function
```

```
result = add(3, 4)
print(result)  # Output: 7
```

Key points:

- Functions can have zero or more parameters.
- Functions can return a value using the `return` statement.
- Functions can be nested within other functions.
- Lambda functions are often used for short, simple functions.

By understanding these different ways to define and call functions, you can write more organized and efficient Python code.

Que_4. What is the purpose of the `return` statement in a Python function?

The `return` statement in Python is used to:

- **Exit a function:** When the `return` statement is executed, the function terminates and control is returned to the calling code.
- **Provide a value:** The `return` statement can optionally return a value to the calling code. This value can be used by the calling code for further processing or calculations.

Example:

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

sum = add(3, 4)
print(sum)  # Output: 7
```

In this example:

- The `add` function takes two arguments, `x` and `y`, and returns their sum.
- The `return` statement is used to return the calculated sum to the calling code.
- The returned value is stored in the `sum` variable, which can then be used for further calculations or output.

Key points:

- A function can have multiple `return` statements, but only the first one executed will be effective.
- If a function doesn't have a `return` statement, it implicitly returns `None`.
- The `return` statement can be used to return any type of value, including numbers, strings, lists, tuples, dictionaries, and objects.

The `return` statement is a fundamental part of Python functions, allowing you to create reusable code that can produce meaningful output.

Que_5. What are iterators in Python and how do they differ from iterables?

Iterators vs. Iterables in Python

Both iterators and iterables are concepts related to iteration, or the process of accessing elements one by one in a collection. However, they have distinct characteristics:

Iterables:

- **Objects that can be iterated over.**
- **Examples:** Lists, tuples, strings, dictionaries, sets.
- **Provide an `__iter__` method:** This method returns an iterator object.

Iterators:

- **Objects that implement the `__iter__` and `__next__` methods.**
- **Used to access elements of an iterable one by one.**
- **Created using the `iter()` function.**

Key Differences:

Feature	**Iterables**	**Iterators**
Definition	Objects that can be iterated over	Objects that implement `__iter__` and `__next__`
Usage	Used directly in `for` loops	Created using `iter()` and used in `for` loops
Methods	Provide `__iter__` method	Provide `__iter__` and `__next__` methods

Example:

```
my_list = [1, 2, 3]  # Iterable

# Creating an iterator from the list
my_iterator = iter(my_list)

# Using the iterator
element1 = next(my_iterator)  # Output: 1
element2 = next(my_iterator)  # Output: 2
element3 = next(my_iterator)  # Output: 3
```

In summary:

- **Iterables** are objects that can be iterated over, such as lists, tuples, and strings.
- **Iterators** are objects that implement the `__iter__` and `__next__` methods, allowing you to access elements one by one.
- Iterators are often created from iterables using the `iter()` function.
- Both iterables and iterators are fundamental concepts in Python for working with collections of elements.

Que_6. Explain the concept of generators in Python and how they are defined.

Generators in Python

Generators are a special type of function that returns an iterator. Unlike regular functions that return a single value, generators return a sequence of values one at a time, using the `yield` keyword. This allows for efficient memory usage, especially when dealing with large datasets.

Defining Generators:

1. **Use the `yield` keyword:** Instead of using `return`, generators use `yield` to return values one by one.
2. **Pause and resume:** When a generator function encounters a `yield` statement, it pauses execution and returns the value. The function's state is saved, allowing it to resume execution from the same point the next time it's called.

Example:

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

for num in count_up(5):
    print(num)
```

Output:

```
0
1
2
3
4
```

Key points about generators:

- **Efficient memory usage:** Generators avoid creating and storing the entire sequence in memory at once.
- **Lazy evaluation:** Elements are generated on-demand, as needed.
- **Can be used in `for` loops:** Generators can be used directly in `for` loops to iterate over their values.
- **Customizable:** You can create custom generators to produce specific sequences of values.

Common use cases for generators:

- **Infinite sequences:** Generating infinite sequences like Fibonacci numbers or prime numbers.
- **Data processing:** Processing large datasets in chunks to avoid memory issues.
- **Custom iterators:** Creating custom iterators for specific use cases.

Generators are a powerful feature in Python that can enhance code efficiency and readability, especially when dealing with large or infinite sequences of data.

Que_7. What are the advantages of using generators over regular functions?

Advantages of Generators Over Regular Functions:

1. **Memory Efficiency:** Generators avoid creating and storing the entire sequence in memory at once. Instead, they produce elements one at a time as needed, which can be especially beneficial when dealing with large datasets.
2. **Lazy Evaluation:** Generators are evaluated lazily, meaning elements are generated only when requested. This can improve performance, especially when dealing with infinite sequences or sequences that are computationally expensive to generate.
3. **Simpler Code:** Generators can often lead to cleaner and more concise code, especially for tasks that involve producing sequences of values.
4. **Custom Iterators:** Generators can be used to create custom iterators, providing more flexibility and control over the iteration process.
5. **Infinite Sequences:** Generators can be used to generate infinite sequences, which are not possible with regular functions.

In summary, generators offer several advantages over regular functions, including improved memory efficiency, lazy evaluation, and the ability to create custom iterators. They are particularly useful for tasks that involve large datasets, infinite sequences, or complex iteration patterns.

Example:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Using the generator to print the first 10 Fibonacci numbers
for num in fibonacci():
    if num > 100:
        break
    print(num)
```

In this example, the `fibonacci` generator function produces an infinite sequence of Fibonacci numbers. By using `yield`, the generator can pause and resume execution, allowing it to generate numbers one at a time. This avoids creating a large list of all Fibonacci numbers upfront, which could be memory-intensive.

By using generators, you can write more efficient and concise code, especially when dealing with large or infinite datasets

Que_8. What is a lambda function in Python and when is it typically used?

Lambda Functions in Python

A lambda function, also known as an anonymous function, is a small, inline function defined using the `lambda` keyword. It's a concise way to create functions without giving them a name.

Syntax:

```
lambda arguments: expression
```

Example:

```
add = lambda x, y: x + y
result = add(3, 4)
print(result)  # Output: 7
```

Typical Uses:

1. **Short, simple functions:** Lambda functions are often used for short, one-line functions that are used only once.
2. **Passing functions as arguments:** They can be passed as arguments to other functions, such as `map`, `filter`, and `reduce`.
3. **Creating anonymous callbacks:** Lambda functions are often used as callback functions in event-driven programming.

Example:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

Key points about lambda functions:

- They are defined using the `lambda` keyword.
- They have no name.
- They can take any number of arguments.
- They can return a single expression.
- They are often used for concise, one-time functions.

Lambda functions are a powerful tool in Python for creating concise and expressive code.

Que_9. Explain the purpose and usage of the `map()` function in Python.

`map()` function in Python

The `map()` function in Python applies a given function to each element of an iterable (like a list, tuple, or string) and returns a new iterable containing the results.

Syntax:

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

Parameters:

- `function`: The function to apply to each element of the iterable.
- `iterable`: The iterable object.

Return value:

- A new iterable containing the results of applying the function to each element of the original iterable.

Example:

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

squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

In this example, the `map()` function applies the lambda function `lambda x: x**2` to each element of the `numbers` list, squaring each number and returning a new list of squared values.

Key points about `map()`:

- It's a convenient way to apply a function to multiple elements of an iterable.
- It returns a new iterable, leaving the original iterable unchanged.
- It can be used with any function that takes a single argument.
- It's often used in conjunction with lambda functions for concise expressions.

Common use cases for `map()`:

- Applying mathematical operations to elements of a list or array.
- Transforming elements of a list or array into a different data type.
- Creating new lists or arrays based on existing ones.

The `map()` function is a powerful tool for functional programming in Python, allowing you to perform operations on multiple elements efficiently and concisely.

Que_10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

`map()`, `reduce()`, and `filter()` are three built-in functions in Python that are commonly used for functional programming. They provide concise and efficient ways to perform operations on iterables.

`map()`:

- **Purpose:** Applies a function to each element of an iterable and returns a new iterable containing the results.
- **Syntax:** `map(function, iterable)`
- **Example:**

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

`reduce()`:

- **Purpose:** Applies a function to an iterable, reducing it to a single value.
- **Syntax:** `reduce(function, iterable, initializer=None)`
- **Example:**

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output:
 120
```

`filter()`:

- **Purpose:** Creates a new iterable containing elements from the original iterable that satisfy a given condition.
- **Syntax:** `filter(function, iterable)`
- **Example:**

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4]
```

Key Differences:

Function	Purpose	Return Value
`map()`	Applies a function to each element	New iterable with results
`reduce()`	Reduces an iterable to a single value	Single value
`filter()`	Filters elements based on a condition	New iterable with filtered elements

In summary:

- **`map()`** is used for applying a function to each element.
- **`reduce()`** is used for combining elements into a single value.
- **`filter()`** is used for selecting elements based on a condition.

Que_11. Using pen & Paper write the internal mechanism for sum operation using reduce function on this givenlist:[47,11,42,13];

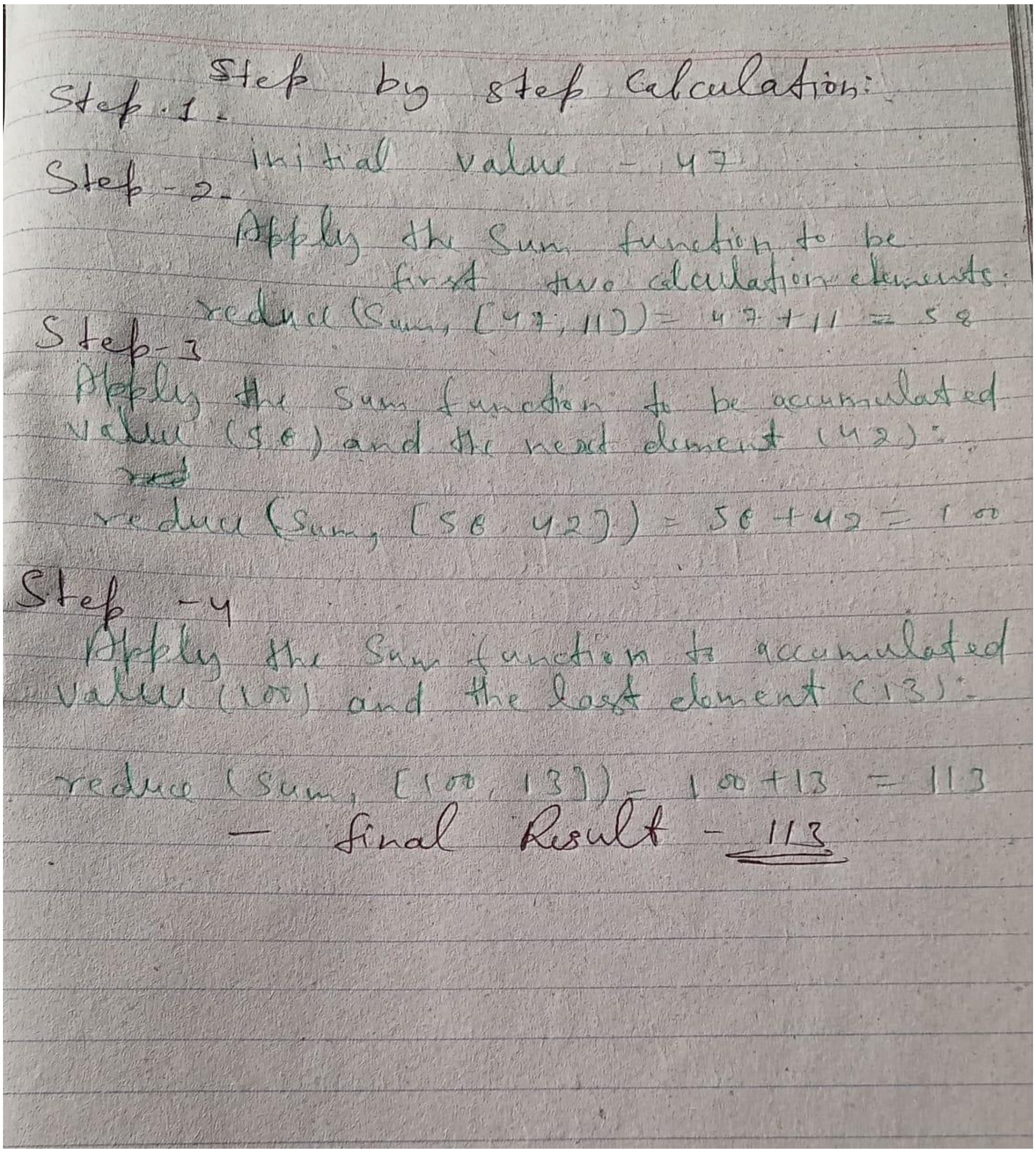
Understanding `reduce()`:

The `reduce()` function in Python takes an iterable (like a list) and a function, and applies the function cumulatively to the elements of the iterable, reducing it to a single value.

Internal Mechanism:

1. **Initialization:**
   - The `reduce()` function starts with an initial value (if provided) or the first element of the iterable. In this case, since no initial value is provided, the first element (47) will be the initial value.
2. **Iteration:**
   - The function iterates over the remaining elements of the list (11, 42, 13).
   - At each step, it applies the provided function to the current element and the accumulated value.
3. **Accumulation:**
   - The result of the function application becomes the new accumulated value for the next iteration.

Step-by-Step Calculation:



Step by step Calculation:

Step.1 -

initial value - 47

Step - 2 -

Apply the Sum function to be first two calculation elements:

reduce (Sum, [47, 11]) = 47 + 11 = 58

Step - 3

Apply the sum function to be accumulated value (58) and the next element (42):

~~red~~

reduce (Sum, [58 42]) = 58 + 42 = 100

Step - 4

Apply the Sum function to accumulated value (100) and the last element (13):

reduce (Sum, [100, 13]) - 100 + 13 = 113

— final Result - 113

Step 1: Initial value = 47

Step 2: Apply the `sum` function to the first two elements:

- `reduce(sum, [47, 11]) = 47 + 11 = 58`

Step 3: Apply the `sum` function to the accumulated value (58) and the next element (42):

- `reduce(sum, [58, 42]) = 58 + 42 = 100`

Step 4: Apply the `sum` function to the accumulated value (100) and the last element (13):

- `reduce(sum, [100, 13]) = 100 + 13 = 113`

Final Result:

The `reduce()` function returns the final accumulated value, which is 113. This is the sum of the elements in the list [47, 11, 42, 13].

In essence, the `reduce()` function iteratively combines the elements of a list using a specified function, producing a single output value.

Practical Questions

Que_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):
  """Calculates the sum of even numbers in a list.


  Args:
    numbers: A list of numbers.


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


  sum_even = 0
  for num in numbers:
    if num % 2 == 0:
      sum_even += num
  return sum_even


# Example usage:
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]

result = sum_even_numbers(numbers)

print(result)  # Output: 20
```

Que_2. Create a Python function that accepts a string and returns the reverse of that string.

```
def reverse_string(string):

  """Reverses a given string.


  Args:

    string: The string to be reversed.


  Returns:

    The reversed string.

  """


  reversed_string = ""

  for char in string:

    reversed_string = char + reversed_string

  return reversed_string
```

This function works by:

1. **Initializing an empty string `reversed_string`** to store the reversed characters.
2. **Iterating over each character in the input string:** For each character, it concatenates the character to the beginning of the `reversed_string`. This effectively reverses the order of the characters.
3. **Returning the reversed string:** After iterating over all the characters, the function returns the `reversed_string`.

You can use this function like this:

```
string = "hello, world!"
reversed_string = reverse_string(string)
print(reversed_string)  # Output: !dlrow ,olleh
```

Que_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):
  """Squares each number in a given list.


  Args:
    numbers: A list of integers.


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


  squared_numbers = []
  for num in numbers:
    squared_numbers.append(num ** 2)
  return squared_numbers
numbers = [1, 2, 3, 4, 5]
squared_numbers = square_numbers(numbers)
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

Que_4. Write a Python function that checks if a given number is prime or not from 1 to 200

```
def is_prime(number):
  """Checks if a given number is prime.
  Args:
    number: The number to check.

  Returns:
    True if the number is prime, False otherwise.
  """

  if number <= 1:
    return False

  for i in range(2, int(number**0.5) + 1):
    if number % i == 0:
      return False

  return True

# Check if a number is prime
num = int(input("Enter a number: "))
if is_prime(num):
  print(f"{num} is a prime number.")
else:
  print(f"{num} is not a prime number.")
```

This function works by:

1. Checking if the number is less than or equal to 1. If it is, it's not prime.

2. Iterating from 2 to the square root of the number. If the number is divisible by any of these numbers, it's not prime.
3. If the number is not divisible by any number from 2 to its square root, it's prime.

This function is efficient because it only needs to check divisibility up to the square root of the number.

Que_5. Create an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms.

```
class FibonacciIterator:

    def __init__(self, terms):

        self.terms = terms

        self.a = 0

        self.b = 1

        self.count = 0


    def __iter__(self):

        return self


    def __next__(self):

        if self.count >= self.terms:

            raise StopIteration

        fib = self.a

        self.a, self.b = self.b, self.a + self.b

        self.count += 1

        return fib
```

Explanation:

1. **Initialization:** The `__init__` method initializes the iterator class with the desired number of terms, the first two Fibonacci numbers (0 and 1), and a counter to keep track of the current term.
2. **`__iter__` method:** This method is required for an object to be iterable. It returns the iterator object itself.
3. **`__next__` method:** This method is called each time the iterator is used in a `for` loop. It generates the next Fibonacci number and returns it. If the desired number of terms has been reached, it raises a `StopIteration` exception to indicate that there are no more elements.

Usage:

```
fib_iterator = FibonacciIterator(10)

for num in fib_iterator:
    print(num)
```

This will print the first 10 Fibonacci numbers:

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

This iterator class effectively generates the Fibonacci sequence up to the specified number of terms, demonstrating the power of iterators in Python.

Que_6. Write a generator function in Python that yields the powers of 2 up to a given exponent.

```
def powers_of_two(exponent):
  """Generates powers of 2 up to a given exponent.

  Args:
    exponent: The maximum exponent.

  Yields:
    Powers of 2 up to the given exponent.
  """

  power = 0
  while power <= exponent:
    yield 2 ** power
    power += 1
```

This generator function works by:

1. **Initializing a variable `power` to 0.**
2. **Using a `while` loop to generate powers of 2:**
   - The `yield` keyword is used to return the current power of 2.
   - The `power` variable is incremented after each iteration.
3. **The loop continues until the `power` exceeds the given `exponent`.**

You can use this generator function like this:

```
exponent = 5
for power in powers_of_two(exponent):
  print(power)

This will print the powers of 2 up to the given exponent:
1
2
4
8
16
32
```

Que_7. Implement a generator function that reads a file line by line and yields each line as a string

Here's a Python generator function that reads a file line by line and yields each line as a string:

```
def read_file_lines(filename):
  """Reads a file line by line and yields each line as a string.

  Args:
    filename: The name of the file to read.

  Yields:
    Each line of the file as a string.
  """

  with open(filename, 'r') as file:
    for line in file:
      yield line.strip()
```

Explanation:

1. **`with open(filename, 'r') as file:`**: This opens the file specified by `filename` in read mode (`'r'`). The `with` statement ensures that the file is automatically closed when the code block is finished, even if an exception occurs.
2. **`for line in file:`**: This iterates over each line in the file.
3. **`yield line.strip()`**: The `yield` keyword is used to return each line of the file as a string. The `strip()` method is used to remove any leading or trailing whitespace from the line.

Usage:

```
filename = "my_file.txt"

for line in read_file_lines(filename):
  print(line)
```

This code will read the contents of the file `my_file.txt` line by line and print each line to the console.

Key points:

- The generator function uses the `yield` keyword to return each line of the file one at a time.
- The `with` statement is used to ensure proper file handling and closing.
- The `strip()` method is used to remove any leading or trailing whitespace from each line.

This generator function can be useful for processing large files efficiently, as it avoids reading the entire file into memory at once.

Que_8. Use a lambda function in Python to sort a list of tuples based on the second element of each tuple.

# Create a list of tuples

my_list = [('apple', 3), ('banana', 2), ('orange', 4), ('grape', 1)]

# Sort the list based on the second element of each tuple

sorted_list = sorted(my_list, key=lambda x: x[1])

# Print the sorted list

print(sorted_list)

Output:

```
[('grape', 1), ('banana', 2), ('apple', 3), ('orange', 4)]
```

Que_9. Write a Python program that uses `map()` to convert a list of temperatures from Celsius to Fahrenheit.

```
def celsius_to_fahrenheit(celsius):
  """Converts Celsius to Fahrenheit.

  Args:
    celsius: The temperature in Celsius.

  Returns:
    The temperature in Fahrenheit.
  """

  return (celsius * 9/5) + 32

# List of temperatures in Celsius
celsius_temperatures = [25, 30, 35, 40]

# Convert Celsius to Fahrenheit using the map function
fahrenheit_temperatures = list(map(celsius_to_fahrenheit, celsius_temperatures))

print(fahrenheit_temperatures)
```

This code will output:

```
[77.0, 86.0, 95.0, 104.0]
```

Que_10. Create a Python program that uses `filter()` to remove all the vowels from a given string

```
def remove_vowels(string):
  """Removes all vowels from a given string.

  Args:
    string: The input string.

  Returns:
    The string with all vowels removed.
  """

  vowels = "aeiouAEIOU"
  return "".join(filter(lambda char: char not in vowels, string))

# Example usage
string = "Hello, World!"
result = remove_vowels(string)
print(result)  # Output: Hll, Wrld!
```

Que_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

```
def process_orders(orders):
  """Processes a list of orders and returns a list of tuples containing the order number and the total
price.

  Args:
    orders: A list of sublists, where each sublist contains the order number, book title, author,
quantity, and price per item.

  Returns:
    A list of tuples, where each tuple contains the order number and the total price.
  """

  processed_orders = []
  for order in orders:
    order_number, _, _, quantity, price_per_item = order
    total_price = quantity * price_per_item
    if total_price < 10000:
      total_price += 10
    processed_orders.append((order_number, total_price))
  return processed_orders

# Example usage
orders = [
  (1001, "Python Programming", "John Doe", 2, 30),
  (1002, "Data Structures", "Jane Smith", 3, 25),
  (1003, "Algorithms", "Mike Johnson", 1, 50),
]

processed_orders = process_orders(orders)
print(processed_orders)
```