ASSIGNMENT FUNCTION

Theory Questions:

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

In Python, both functions and methods are blocks of reusable code designed to perform specific tasks, but their key difference lies in their association with objects and classes.

Function:

A function is a standalone block of code that is not associated with any specific object or class. It can be defined and called independently. Functions operate on the data passed to them as arguments and can return a value.

Example of a Function:

Python

```
def greet(name):
  """This function greets the person passed in as an argument."""
  return f"Hello, {name}!"

message = greet("Alice")
print(message)
```

Method:

A method is a function that belongs to a class and is called on an instance (object) of that class. Methods are implicitly passed the object on which they are called (often referred to as `self` within the method definition), allowing them to access and potentially modify the object's data (attributes).

Example of a Method:

Python

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

  def bark(self):
    """This method makes the dog bark."""
    return f"{self.name} says Woof!"

my_dog = Dog("Buddy", "Golden Retriever")
print(my_dog.bark())
```

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

In Python, parameters are the variables listed inside the parentheses in the function definition. They act as placeholders for the values that the function expects to receive when it is called. Arguments are the actual values passed to the function when it is invoked.

Here's a breakdown with an example: Parameters (Function Definition).

When defining a function, you specify parameters to indicate what kind of input the function needs. These are essentially local variables within the function's scope.
Arguments (Function Call).
When you call a function, you provide arguments, which are the concrete values that get assigned to the corresponding parameters.
Example:
Python

```
# Function definition with parameters 'name' and 'age'
def introduce_yourself(name, age):
  print(f"Hello, my name is {name} and I am {age} years old.")

# Function call with arguments "Alice" and 30
introduce_yourself("Alice", 30)

# Another function call with different arguments "Bob" and 25
introduce_yourself("Bob", 25)
```

Explanation:

- **In the `def introduce_yourself(name, age):` line, `name` and `age` are the parameters. They are placeholders for the information the function needs to perform its task.**
- **In the line `introduce_yourself("Alice", 30)`, `"Alice"` and `30` are the arguments. When this line executes, `"Alice"` is passed as the argument for the `name` parameter, and `30` is passed as the argument for the `age` parameter.**
- **Similarly, in `introduce_yourself("Bob", 25)`, `"Bob"` and `25` are the arguments for `name` and `age` respectively.**

Q3. What are the different ways to define and call a function in Python?
In Python, functions are defined using the `def` keyword and called by using their name followed by parentheses.
Defining a Function:
Basic Function Definition.
A function is defined using the `def` keyword, followed by the function name, parentheses `()`, and a colon `:`. The function's code block is then indented.
Python

```
  def greet():
      print("Hello, world!")
```

function with parameters.
Parameters are variables listed inside the parentheses during function definition. These parameters act as placeholders for values that will be passed into the function when it is called.
Python

```
  def greet_user(name):
      print(f"Hello, {name}!")
```

Function with Return Value.

Functions can return values using the `return` statement. This allows the function's result to be used in other parts of the program.

Python

```
def add(a, b):
    return a + b
```

Calling a Function:

Calling a Basic Function.

To execute the code within a function, simply type the function's name followed by parentheses `()`.

Python

```
greet()  # Output: Hello, world!
```

Calling a Function with Arguments.

When calling a function that expects parameters, provide the actual values (arguments) inside the parentheses.

Python

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

- **Calling a Function and Storing the Return Value:**

If a function returns a value, you can store it in a variable or use it directly.

Python

```
result = add(5, 3)
print(result)  # Output: 8
```

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

The `return` statement in a Python function serves two primary purposes:

- **To exit the function: When a `return` statement is encountered during the execution of a function, the function immediately terminates, and control is passed back to the point in the code where the function was called. Any code written after the `return` statement within the function will not be executed.**
- **To send a value back to the caller: The `return` statement can optionally be followed by a value or an expression. This value is then sent back as the result of the function call to the part of the program that invoked the function. This allows functions to compute results and provide them for further use in the program. If no value is specified after `return`, or if the function completes without encountering a `return` statement, it implicitly returns `None`.**

Here is an example demonstrating the use of the `return` statement:

Python

```
def calculate_area(length, width):
    """
    Calculates the area of a rectangle.
    """
    area = length * width
    return area  # Returns the calculated area
```

```
def greet(name):
    """
    Prints a greeting message.
    This function does not explicitly return a value.
    """
    print(f"Hello, {name}!")

# Calling functions and utilizing return values
rectangle_length = 10
rectangle_width = 5
result_area = calculate_area(rectangle_length, rectangle_width)
print(f"The area of the rectangle is: {result_area}")

# Calling a function that does not explicitly return a value
greet("Alice")
returned_value_from_greet = greet("Bob") # This will be None
print(f"The value returned by greet('Bob') is: {returned_value_from_greet}")
```

In this example:

- `calculate_area` uses `return area` to send the computed area back to the caller, which then stores it in `result_area`.
- `greet` does not have an explicit `return` statement. When called, it prints a message, but when its result is assigned to `returned_value_from_greet`, the value is `None` because no explicit value was returned.

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

In Python, iterables are objects that can be iterated over, meaning you can loop through their elements one by one. Examples include lists, tuples, strings, and dictionaries. An object is iterable if it implements the `__iter__()` method, which returns an iterator, or the `__getitem__()` method, which allows access to elements by index.

Iterators, on the other hand, are objects that represent a stream of data and provide a way to access elements sequentially. An iterator is an object that implements both the `__iter__()` method (which returns itself) and the `__next__()` method. The `__next__()` method returns the next item in the sequence, and raises a `StopIteration` exception when there are no more items.

Key Differences:

- Capabilities: Iterables can be iterated over, while iterators are the objects that perform the iteration.
- Methods: Iterables typically implement `__iter__()` (or `__getitem__()`), while iterators implement both `__iter__()` and `__next__()`.
- State: Iterators maintain an internal state to keep track of the current position in the sequence, allowing them to deliver elements one at a time. Iterables do not inherently maintain this state.

- Creation: You can obtain an iterator from an iterable using the built-in `iter()` function.
-

Example:
Python

```
# An iterable (a list)
my_list = [10, 20, 30]

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

# Use the iterator to get elements one by one
print(next(my_iterator))  # Output: 10
print(next(my_iterator))  # Output: 20
print(next(my_iterator))  # Output: 30

# Trying to get another element will raise StopIteration
try:
  print(next(my_iterator))
except StopIteration:
  print("No more elements in the iterator.")

# You can iterate directly over the iterable using a for loop (which implicitly uses iterators)
print("\nIterating with a for loop:")
for item in my_list:
  print(item)
```

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

Python generators are a type of iterator that allow for the generation of values on the fly, one at a time, without storing the entire sequence in memory. This makes them highly memory-efficient, especially when dealing with large datasets or potentially infinite sequences.

Definition:

Generators are defined like regular Python functions, but instead of using the `return` keyword to return a value and terminate the function, they use the `yield` keyword. When `yield` is encountered, the function's execution is paused, the yielded value is returned, and the function's state (including local variables) is saved. When the generator is called again (e.g., in a `for` loop or by calling `next()`), execution resumes from where it left off.

Example:
Python

```
def count_up_to(n):
  """
  A generator function that yields numbers from 0 up to (but not including) n.
  """
```

```
    i = 0
    while i < n:
        yield i  # Yields the current value of i
        i += 1

# Creating a generator object
my_generator = count_up_to(5)

# Iterating through the generator
print("Using a for loop:")
for num in my_generator:
    print(num)

# Re-creating the generator to demonstrate next()
my_generator = count_up_to(3)

print("\nUsing next():")
print(next(my_generator))  # Output: 0
print(next(my_generator))  # Output: 1
print(next(my_generator))  # Output: 2

# Attempting to get the next value after exhaustion will raise StopIteration
try:
    print(next(my_generator))
except StopIteration:
    print("Generator exhausted.")
```

In this example, `count_up_to(n)` is a generator function. When `count_up_to(5)` is called, it returns a generator object. When this object is iterated over (either with a `for` loop or `next()`), the `yield i` statement provides values one by one, pausing and resuming execution as needed. This allows for efficient processing of sequences without consuming excessive memory.

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

Generators offer several advantages over regular functions, primarily related to memory efficiency and handling of large or infinite sequences.

Advantages of Generators:

- Memory Efficiency: Generators produce values one at a time and do not store the entire sequence in memory. This is crucial when dealing with very large datasets or infinite sequences, as it prevents memory exhaustion. Regular functions, when returning a list or similar data structure, store all elements in memory simultaneously.
- Lazy Evaluation: Values are generated on demand, only when requested by the iteration process. This means that computation for subsequent values is deferred until needed, which can save computational resources if not all values are ultimately consumed.

- **Handling Infinite Sequences: Generators can easily represent and work with infinite sequences, as they don't need to create the entire sequence in memory. Regular functions cannot directly handle infinite sequences in this way.**
- **Cleaner Code for Iterators: Generators provide a more concise and readable way to write iterators compared to implementing `__iter__` and `__next__` methods in a class.**

Example:
Consider generating a sequence of even numbers up to a certain limit.
Using a Regular Function:
Python

```
def get_even_numbers_list(n):
  even_numbers = []
  for i in range(n + 1):
    if i % 2 == 0:
      even_numbers.append(i)
  return even_numbers

# This creates a list in memory containing all even numbers
all_evens = get_even_numbers_list(1000000)
# print(all_evens) # Not recommended for large n
```

Using a Generator Function:
Python

```
def get_even_numbers_generator(n):
  for i in range(n + 1):
    if i % 2 == 0:
      yield i

# This creates a generator object, not a list
even_gen = get_even_numbers_generator(1000000)

# Values are generated one by one as we iterate
for number in even_gen:
  if number > 10: # Example of consuming only a portion
    break
  print(number)
```

In the generator example, `even_gen` does not store all million even numbers in memory. Instead, it yields each even number as it's requested by the `for` loop, making it significantly more memory-efficient for large `n`.

Q8. What is a lambda function in Python and when is it typically used?
A lambda function in Python is a small, anonymous function defined using the `lambda` keyword. Unlike regular functions defined with `def`, lambda functions do not require a name and are limited to a single expression, which is implicitly returned.
Syntax:
Python

```
lambda arguments: expression
```

- `arguments`: These are the input parameters to the lambda function, similar to function arguments in a `def` function.
- `expression`: This is a single expression that is evaluated and returned by the lambda function.

When is it typically used?
Lambda functions are typically used for:

- **Short, one-time use functions: When you need a simple function for a brief period and defining a full `def` function would be excessive.**
- **With higher-order functions: They are frequently used as arguments to higher-order functions like `map()`, `filter()`, `sorted()`, and `reduce()`, which take functions as arguments.**
- **Concise code: They can make your code more concise and readable for simple operations.**

Example:
Consider a list of numbers that needs to be sorted based on their absolute values.
Python

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

# Sorting using a lambda function as the key for sorted()
sorted_numbers = sorted(numbers, key=lambda x: abs(x))

print(f"Original numbers: {numbers}")
print(f"Sorted by absolute value: {sorted_numbers}")

# Another example: using map to double each number
doubled_numbers = list(map(lambda x: x * 2, numbers))
print(f"Doubled numbers: {doubled_numbers}")
```

In this example:

- `lambda x: abs(x)` **creates an anonymous function that takes an argument `x` and returns its absolute value. This is passed to the `key` parameter of `sorted()`, which uses it to determine the sorting order.**
- `lambda x: x * 2` **creates an anonymous function that doubles its input, which is then applied to each element in `numbers` by the `map()` function.**

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

The `map()` function in Python serves the purpose of applying a specified function to each item in an iterable (such as a list, tuple, or set) and returning a `map` object, which is an iterator containing the results. This function provides a concise and efficient way to perform transformations on collections of data without explicitly writing a `for` loop.
Usage:

The `map()` function takes two main arguments:

- **`function`: The function to be applied to each item of the iterable(s). This can be a built-in function, a user-defined function, or a lambda function.**
- **`iterable`: One or more iterables whose elements will be passed as arguments to the `function`. If multiple iterables are provided, the function must accept a corresponding number of arguments.**

Example:
Consider the task of squaring each number in a list.
Python

```
# Define a list of numbers
numbers = [1, 2, 3, 4, 5]

# Define a function to square a number
def square(num):
  return num * num

# Use map() to apply the square function to each number
squared_numbers_map = map(square, numbers)

# Convert the map object to a list to view the results
squared_numbers_list = list(squared_numbers_map)

print(f"Original numbers: {numbers}")
print(f"Squared numbers: {squared_numbers_list}")

# Example using a lambda function for a more concise approach
cubed_numbers_list = list(map(lambda x: x**3, numbers))
print(f"Cubed numbers (using lambda): {cubed_numbers_list}")
```

In this example, `map(square, numbers)` applies the `square` function to each element in the `numbers` list. The result is a `map` object, which is then converted to a list to display the transformed values. The second example demonstrates using a `lambda` function directly within `map()` for a more compact solution.

Q10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

The `map()`, `filter()`, and `reduce()` functions in Python are higher-order functions used for processing iterables, each serving a distinct purpose:

- **`map()`:**
  - **Purpose: Applies a given function to each item in an iterable and returns an iterator that yields the results. It transforms elements.**
  - **Output: An iterator (which can be converted to a list, tuple, etc.) with the same number of elements as the input iterable.**
  - **Example:**

Python

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

- **`filter()`:**
  - **Purpose: Constructs an iterator from elements of an iterable for which a function returns `True`. It selects elements based on a condition.**
  - **Output: An iterator (which can be converted to a list, tuple, etc.) containing only the elements that satisfy the condition. The number of elements can be less than or equal to the input iterable.**
  - **Example:**

Python

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
# Output: [2, 4, 6]
```

- **`reduce()`:**
  - **Purpose: Applies a function of two arguments cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single value. It aggregates elements.**
  - **Requirement: Must be imported from the `functools` module.**
  - **Output: A single value.**
  - **Example:**

Python

```
from functools import reduce

numbers = [1, 2, 3, 4]
sum_of_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_of_numbers)
# Output: 10
```

In summary: map() transforms each element, filter() selects elements based on a condition, and reduce() aggregates elements into a single result.

Q11. . Using pen & Paper write the internal mechanism for sum operation using reduce function on this given List:[47,11,42,13];?

The `reduce` function, often found in functional programming paradigms, iteratively applies a given function to the elements of a list, reducing it to a single cumulative value. When used for summation, it works as follows:

Internal Mechanism for Sum Operation using `reduce`:

- **Initialization: The `reduce` function starts with an initial "accumulator" value. If an initial value is not explicitly provided, it defaults to the first element of the list.**
- **Iteration: The `reduce` function then iterates through the remaining elements of the list, one by one.**
- **Application of Function: In each iteration, it takes the current "accumulator" value and the current element from the list, and applies the provided function (in this case, an addition function) to them.**
- **Update Accumulator: The result of this function application becomes the new "accumulator" value for the next iteration.**
- **Final Result: This process continues until all elements in the list have been processed. The final "accumulator" value is the result of the `reduce` operation.**

Let's assume the `reduce` function is called with a lambda function `lambda accumulator, current_element: accumulator + current_element` and an initial accumulator value of `0`.

- **Step 1:**
  - **`accumulator` = `0` (initial value)**
  - **`current_element` = `47` (first element of the list)**
  - **Apply `0 + 47 = 47`.**
  - **New `accumulator` = `47`.**
- **Step 2:**
  - **`accumulator` = `47`**
  - **`current_element` = `11`**
  - **Apply `47 + 11 = 58`.**
  - **New `accumulator` = `58`.**
- **Step 3:**
  - **`accumulator` = `58`**
  - **`current_element` = `42`**
  - **Apply `58 + 42 = 100`.**
  - **New `accumulator` = `100`.**
- **Step 4:**
  - **`accumulator` = `100`**
  - **`current_element` = `13`**
  - **Apply `100 + 13 = 113`.**
  - **New `accumulator` = `113`.**

Final Result: After processing all elements, the `reduce` function returns the final `accumulator` value, which is `113`.

Practical Questions:

Q1. Write a Python function that takes a list of numbers as input and returns the sum of all even numbers in the list.?

Here is a Python function that calculates the sum of all even numbers in a given list:

Python

```
def sum_even_numbers(numbers):
 """
  Calculates the sum of all even numbers in a list.

  Args:
    numbers: A list of integers.

  Returns:
    The sum of all even numbers in the list.
 """
 total_sum = 0
 for number in numbers:
   if number % 2 == 0:  # Check if the number is even
     total_sum += number
 return total_sum
```

Q2. Create a Python function that accepts a string and returns the reverse of that string?

Here is a Python function that accepts a string and returns its reverse:

Python

```
def reverse_string(input_string):
 """
  Reverses a given string.

  Args:
    input_string: The string to be reversed.

  Returns:
    The reversed string.
 """
 return input_string[::-1]
```

Q3. Implement a Python function that takes a list of integers and returns a new list containing the squares of each number.?

Here is a Python function that takes a list of integers and returns a new list containing the squares of each number:

Python

```
def square_list_elements(input_list):
```

```
 """
  Takes a list of integers and returns a new list containing the squares of each number.

  Args:
    input_list: A list of integers.

  Returns:
    A new list where each element is the square of the corresponding element in the
input_list.
  """
 squared_list = [num ** 2 for num in input_list]
 return squared_list
```

Q4. Write a Python function that checks if a given number is prime or not from 1 to 200?

A Python function to check if a given number within the range of 1 to 200 is prime can be implemented as follows:

Python

```
import math

def is_prime(number):
  """
   Checks if a given number is prime.

   Args:
      number (int): The number to be checked for primality.

   Returns:
      bool: True if the number is prime, False otherwise.
   """
  if not (1 <= number <= 200):
     raise ValueError("Number must be between 1 and 200 (inclusive).")

  if number <= 1:
     return False  # Numbers less than or equal to 1 are not prime
  if number == 2:
     return True   # 2 is the only even prime number
  if number % 2 == 0:
     return False  # Other even numbers are not prime

  # Check for divisibility by odd numbers from 3 up to the square root of the number
  for i in range(3, int(math.sqrt(number)) + 1, 2):
     if number % i == 0:
        return False  # Found a divisor, so it's not prime
  return True  # No divisors found, so it's prime
```

Q5. Create an iterator class in Python that generates the Fibonacci sequence up to a specified number of

Terms.?

Here is an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms:

Python

```
class FibonacciIterator:
  def __init__(self, num_terms):
    if not isinstance(num_terms, int) or num_terms < 0:
      raise ValueError("Number of terms must be a non-negative integer.")
    self.num_terms = num_terms
    self.current_term_count = 0
    self.a = 0
    self.b = 1

  def __iter__(self):
    return self

  def __next__(self):
    if self.current_term_count < self.num_terms:
      if self.current_term_count == 0:
        self.current_term_count += 1
        return self.a
      elif self.current_term_count == 1:
        self.current_term_count += 1
        return self.b
      else:
        next_fib = self.a + self.b
        self.a = self.b
        self.b = next_fib
        self.current_term_count += 1
        return next_fib
    else:
      raise StopIteration

# Example usage:
fib_sequence = FibonacciIterator(10)
print("Fibonacci sequence up to 10 terms:")
for num in fib_sequence:
  print(num, end=" ")
print()

fib_sequence_five = FibonacciIterator(5)
print("Fibonacci sequence up to 5 terms:")
for num in fib_sequence_five:
  print(num, end=" ")
print()

# Example with zero terms
fib_sequence_zero = FibonacciIterator(0)
```

```
print("Fibonacci sequence up to 0 terms:")
for num in fib_sequence_zero:
  print(num, end=" ")
print()
```

Q6. Write a generator function in Python that yields the powers of 2 up to a given exponent?

A Python generator function that yields the powers of 2 up to a given exponent can be implemented as follows:

Python

```
def powers_of_two_generator(max_exponent):
  """
   Yields powers of 2 from 2^0 up to 2^max_exponent.

   Args:
      max_exponent (int): The highest exponent to calculate the power of 2 for.
                  Must be a non-negative integer.
   """
  if not isinstance(max_exponent, int) or max_exponent < 0:
     raise ValueError("max_exponent must be a non-negative integer.")

  for exponent in range(max_exponent + 1):
     yield 2 ** exponent
```

Explanation:

- `def powers_of_two_generator(max_exponent):`: This defines a function named `powers_of_two_generator` that takes one argument, `max_exponent`.
- `if not isinstance(max_exponent, int) or max_exponent < 0:`: This performs input validation, ensuring that `max_exponent` is a non-negative integer. If not, it raises a `ValueError`.
- `for exponent in range(max_exponent + 1):`: This loop iterates through the exponents, starting from 0 up to `max_exponent` (inclusive). The `range(n)` function generates numbers from 0 up to `n-1`, so `range(max_exponent + 1)` is used to include `max_exponent`.
- **`yield 2 ** exponent`**: This is the core of the generator function. Instead of `return`, `yield` is used. When `yield` is encountered, the function pauses, returns the current value (2 raised to the power of `exponent`), and remembers its state. When the generator is called again (e.g., in a `for` loop), it resumes from where it left off.

Usage Example:

Python

```
# Create a generator object
two_powers = powers_of_two_generator(5)

# Iterate and print the powers of 2
```

```
print("Powers of 2 up to exponent 5:")
for power in two_powers:
  print(power)

# Output:
# Powers of 2 up to exponent 5:
# 1
# 2
# 4
# 8
# 16
# 32
```

Q7. Implement a generator function that reads a file line by line and yields each line as a string.?

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

Python

```
def read_file_lines(filepath):
  """
   A generator function that reads a file line by line and yields each line as a string.

   Args:
      filepath (str): The path to the file to be read.

   Yields:
      str: Each line from the file, including the newline character if present.
   """
  try:
     with open(filepath, 'r', encoding='utf-8') as file:
        for line in file:
           yield line
  except FileNotFoundError:
     print(f"Error: The file '{filepath}' was not found.")
  except Exception as e:
     print(f"An error occurred: {e}")

# Example usage:
if __name__ == "__main__":
  # Create a dummy file for demonstration
  with open("sample.txt", "w") as f:
     f.write("This is line 1.\n")
     f.write("This is line 2.\n")
     f.write("This is line 3.\n")

  print("Reading 'sample.txt' using the generator:")
  for file_line in read_file_lines("sample.txt"):
```

```
    print(file_line.strip())  # .strip() removes leading/trailing whitespace, including
newline

  print("\nAttempting to read a non-existent file:")
  for file_line in read_file_lines("non_existent_file.txt"):
    print(file_line.strip())
```

Q8. Use a lambda function in Python to sort a list of tuples based on the second element of each tuple.?

To sort a list of tuples based on the second element of each tuple in Python, one can use the `sorted()` function or the `.sort()` method with a lambda function as the `key` argument.

Using `sorted()`:

The `sorted()` function returns a new sorted list, leaving the original list unchanged.

Python

```
data = [('apple', 3), ('banana', 1), ('cherry', 2), ('date', 4)]
sorted_data = sorted(data, key=lambda item: item[1])
print(sorted_data)
```

Using the `.sort()` method:

The `.sort()` method sorts the list in-place, modifying the original list directly.

Python

```
data = [('apple', 3), ('banana', 1), ('cherry', 2), ('date', 4)]
data.sort(key=lambda item: item[1])
print(data)
```

In both examples, `lambda item: item[1]` is the lambda function. It takes a single argument `item` (representing a tuple from the list) and returns `item[1]`, which is the second element of the tuple. This returned value is then used by `sorted()` or `.sort()` as the key for comparison during the sorting process.

Q9. Write a Python program that uses `map()` to convert a list of temperatures from Celsius to Fahrenheit.?

Here is a Python program that uses `map()` to convert a list of temperatures from Celsius to Fahrenheit:

Python

```
def celsius_to_fahrenheit(celsius):
 """Converts a temperature from Celsius to Fahrenheit."""
 return (celsius * 9/5) + 32

# List of temperatures in Celsius
celsius_temperatures = [0, 10, 20, 30, 40, 100]

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

# Convert the map object to a list for printing
fahrenheit_list =list(fahrenheit_temperatures)
```

```
print(f"Celsius temperatures: {celsius_temperatures}")
print(f"Fahrenheit temperatures: {fahrenheit_list}")
```

Q10. Create a Python program that uses `filter()` to remove all the vowels from a given string?

Kicking off 6 searches
Looking at 100 sites
Putting it all together

To remove all vowels from a string using the `filter()` function in Python, you can combine `filter()` with a `lambda` function and the `join()` method. The `filter()` function constructs an iterator from elements of an iterable for which a function returns true.

Python program

This program defines a function `remove_vowels()` that takes a string as an argument and returns a new string with all vowels removed.

python

```
def remove_vowels(input_string):
    """
    Removes all vowels (case-insensitive) from a given string using filter().
    """
    vowels = "aeiouAEIOU"

    # Use a lambda function with filter() to keep characters that are NOT vowels
    filtered_characters = filter(lambda char: char not in vowels, input_string)

    # Join the filtered characters to form the new string
    return "".join(filtered_characters)

# Example usage:
my_string = "Hello, World! This is a test."
vowel_free_string = remove_vowels(my_string)

print(f"Original string: '{my_string}'")
print(f"String without vowels: '{vowel_free_string}'")

# Another example
my_string_2 = "The quick brown fox jumps over the lazy dog."
vowel_free_string_2 = remove_vowels(my_string_2)

print(f"\nOriginal string: '{my_string_2}'")
print(f"String without vowels: '{vowel_free_string_2}'")
```

Q11. Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:

ORDER NUMBER BOOKETITTLEANDAUTHOR QUANTITY PRICEPERITEM