Theory Questions

1. **Difference between a function and a method in Python:**

   - **Function:** A function is a block of reusable code that is independent of any object and can be called with zero or more arguments. Functions can perform operations, return results, and be used in various parts of a program. For example, a function to calculate the factorial of a number:

In [ ]:
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))  # Outputs 120
```

```
120
```

   - **Method:** A method is similar to a function but is associated with an object. It can access and modify the object's state. Methods are defined within a class. For instance, consider a `Circle` class with a method to calculate its area:

In [ ]:
```
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * (self.radius ** 2)

circle = Circle(5)
print(circle.area())  # Outputs 78.53981633974483
```

```
78.53981633974483
```

2. **Function arguments and parameters in Python:**

   - **Parameters:** Variables listed inside the parentheses in the function definition. They act as placeholders for the values that will be passed to the function.
   - **Arguments:** Actual values that are passed to the function when it is called.

In [ ]:
```
#Example
def greet(name):  # 'name' is a parameter
    return f"Hello, {name}!"

print(greet("Gaurav"))  # 'Gaurav' is an argument
```

```
Hello, Gaurav!
```

Real-life analogy: Consider a coffee machine (function). The machine has a slot for inserting coins (parameters). When you insert a coin (argument), the machine dispenses

coffee.

3. **Different ways to define and call a function in Python:**

   - **Standard function definition:**

In [ ]:
```
def multiply(x, y):
    return x * y

print(multiply(2, 3))  # Outputs 6
```

```
6
```

Use case: Multiplying two numbers provided by a user in a calculator application.

   - **Lambda function:** A small anonymous function defined using the `lambda` keyword, often used for short-term operations.

In [ ]:
```
multiply = lambda x, y: x * y
print(multiply(2, 3))  # Outputs 6
```

```
6
```

Use case: Sorting a list of dictionaries by a specific key value.

In [ ]:
```
products = [{'name': 'apple', 'price': 10}, {'name': 'banana', 'price': 5}]
sorted_products = sorted(products, key=lambda x: x['price'])

print(sorted_products)
```

```
[{'name': 'banana', 'price': 5}, {'name': 'apple', 'price': 10}]
```

4. **Purpose of the `return` statement in a Python function:**

   The `return` statement terminates the function execution and optionally sends back a value to the caller. It is essential for retrieving the result of a function's computation.

In [ ]:
```
def square(x):
    return x * x
result = square(4)

print(result)  # Outputs 16
```

```
16
```

Creative example: A vending machine function where the return statement represents the item being dispensed.

In [ ]:
```
def dispense_item(code):
    items = {'A1': 'Soda', 'B2': 'Chips', 'C3': 'Candy'}
    return items.get(code, 'Invalid code')

print(dispense_item('B2'))  # Outputs 'Chips'
```

```
Chips
```

5. **Iterators in Python and how they differ from iterables:**

   - **Iterable:** An object capable of returning its members one at a time. Examples include lists, tuples, and strings. Any object with an `__iter__()` method is iterable.
   - **Iterator:** An object that represents a stream of data; it implements the `__next__()` method to fetch the next item. Created using the `iter()` function.

     Real-life analogy: A playlist (iterable) where you can go to the next song using a "next" button (iterator).

In [ ]:
```
my_list = [1, 2, 3]
my_iterator = iter(my_list)
print(next(my_iterator))  # Outputs 1
print(next(my_iterator))  # Outputs 2
print(next(my_iterator))  # Outputs 3
```

```
1
2
3
```

6. **Generators in Python and how they are defined:**

   Generators are special iterators that allow you to declare a function that behaves like an iterator. They are defined using the `yield` keyword and provide a convenient way to implement lazy evaluation.

In [ ]:
```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for number in countdown(5):
    print(number)  # Outputs 5 4 3 2 1
```

```
5
4
3
2
1
```

Creative use case: Simulating a traffic light sequence.

In [ ]:
```
def traffic_light():
    while True:
        yield 'Red'
        yield 'Green'
        yield 'Yellow'

light = traffic_light()
for _ in range(6):
    print(next(light))  # Outputs Red Green Yellow Red Green Yellow
```

```
Red
Green
Yellow
Red
Green
Yellow
```

7. **Advantages of using generators over regular functions:**

   - **Memory Efficiency:** Generators produce items one at a time and only when needed, which is more memory efficient than storing the entire sequence in memory.
   - **Lazy Evaluation:** Values are generated on-the-fly, allowing for handling of large data sets or infinite sequences.
   - **Improved Performance:** Generators can be faster for certain tasks due to reduced memory usage and immediate computation.

   Example: Reading a large file line by line without loading the entire file into memory.

In [ ]:
```
file_path = 'large_file.txt'

# Create a large file with 1000 lines
with open(file_path, 'w') as file:
    for i in range(10):
        file.write(f"This is line number {i}\n")

def read_large_file(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()

for line in read_large_file('large_file.txt'):
    print(line)
```

```
This is line number 0
This is line number 1
This is line number 2
This is line number 3
This is line number 4
This is line number 5
This is line number 6
This is line number 7
This is line number 8
This is line number 9
```

8. **Lambda function in Python and when it is typically used:**

   A lambda function is a small anonymous function defined with the `lambda` keyword. It is used for creating small, one-off functions, typically for short-term use.

In [ ]:
```
add = lambda x, y: x + y
print(add(5, 3))  # Outputs 8
```

```
8
```

Creative example: Filtering a list of names based on a condition.

In [ ]:
```
names = ["Alice", "Bob", "Charlie", "David"]
filtered_names = filter(lambda name: len(name) <= 4, names)
print(list(filtered_names))  # Outputs ['Bob']
```

```
['Bob']
```

9. **Purpose and usage of the `map()` function in Python:**

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

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

Use case: Converting a list of temperatures from Celsius to Fahrenheit.

In [ ]:
```
celsius = [0, 20, 30, 100]
fahrenheit = map(lambda c: (c * 9/5) + 32, celsius)
print(list(fahrenheit))  # Outputs [32.0, 68.0, 86.0, 212.0]
```

```
[32.0, 68.0, 86.0, 212.0]
```

10. **Difference between `map()`, `reduce()`, and `filter()` functions in Python:**

    - **`map()`:** Applies a function to all items in an iterable and returns a map object.

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

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

```
- **`reduce()`:** Applies a rolling computation to sequential
pairs of values in an iterable. It is part of the `functools`
module.
```

In [ ]:
```
from functools import reduce
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Outputs 24
```

```
24
```

- **`filter()`:** Filters items out of an iterable based on a function that returns True or False.

In [ ]:
```
numbers = [1, 2, 3, 4]
evens = filter(lambda x: x % 2 == 0, numbers)
print(list(evens))  # Outputs [2, 4]
```

```
[2, 4]
```

11. **Internal mechanism for sum operation using reduce function on the list `[47,11,42,13]`:**

    (Attach a scanned image or drawing of this) Here is a step-by-step process of how `reduce` works with the sum operation:

    - Initial list: `[47, 11, 42, 13]`
    - Initial call: `47 + 11 = 58`
    - Next call: `58 + 42 =

100 `- Final call:` 100 + 13 = 113 `The final result is` 113`.

Practical Questions

1. **Python function to sum all even numbers in a list:**

In [ ]:
```
def sum_even_numbers(numbers):
       return sum(x for x in numbers if x % 2 == 0)

# Example
numbers = [1, 2, 3, 4, 5, 6]
print(sum_even_numbers(numbers))  # Outputs 12
```

```
12
```

Creative use case: Calculating the sum of even transaction amounts in a financial application to analyze even-numbered transactions.

2. **Python function to reverse a string:**

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

# Example
print(reverse_string("hello"))  # Outputs "olleh"
```

```
olleh
```

Real-life example: Reversing a user's input to create a unique identifier or code.

3. **Python function to return a list of squares of each number:**

In [ ]:
```python
def square_numbers(numbers):
    return [x ** 2 for x in numbers]

# Example
numbers = [1, 2, 3, 4]
print(square_numbers(numbers))  # Outputs [1, 4, 9, 16]
```

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

Use case: Calculating the area of squares given the lengths of their sides.

4. **Python function to check if a number is prime from 1 to 200:**

In [ ]:
```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

# Example
primes = [x for x in range(1, 201) if is_prime(x)]
print(primes)
```

```
[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]
```

Creative example: Checking prime numbers to optimize resource allocation in a distributed computing environment where prime-numbered nodes have special roles.

5. **Iterator class generating the Fibonacci sequence:**

In [ ]:
```python
class Fibonacci:
    def __init__(self, terms):
        self.terms = terms
        self.current = 0
        self.next = 1
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.terms:
            raise StopIteration
        self.count += 1
        fib = self.current
        self.current, self.next = self.next, self.current + self.next
        return fib

# Example
fib_sequence = Fibonacci(10)
for num in fib_sequence:
    print(num, end=" ")  # Outputs 0 1 1 2 3 5 8 13 21 34
```

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

Use case: Generating the Fibonacci sequence for algorithmic training or in financial models to predict growth patterns.

6. **Generator function that yields powers of 2 up to a given exponent:**

In [ ]:
```python
def powers_of_two(exponent):
    for i in range(exponent + 1):
        yield 2 ** i

# Example
for power in powers_of_two(5):
    print(power)  # Outputs 1 2 4 8 16 32
```

```
1
2
4
8
16
32
```

Use case: Generating powers of 2 for network design or cryptographic algorithms.

7. **Generator function that reads a file line by line:**

In [ ]:
```python
def read_file_line_by_line(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()

# Example
for line in read_file_line_by_line('large_file.txt'):
    print(line)
```

```
This is line number 0
This is line number 1
This is line number 2
This is line number 3
This is line number 4
This is line number 5
This is line number 6
This is line number 7
This is line number 8
This is line number 9
```

Real-life application: Efficiently processing log files in a monitoring system.

8. **Lambda function to sort a list of tuples based on the second element:**

In [ ]:
```python
# Original tuples
tuples = [(1, 3), (4, 1), (5, 2)]

# Sorting function with explanation
def sort_by_second(data):
    """
```

```
    Sorts a list of data elements based on their second element.

    Args:
        data: A list of tuples or similar data structures with two elements.

    Returns:
        A new list containing the elements sorted by their second element.
    """
    return sorted(data, key=lambda x: x[1])

# Sort the tuples using the function
sorted_tuples = sort_by_second(tuples.copy())

print(sorted_tuples)  # Output: [(4, 1), (5, 2), (1, 3)]
```

```
[(4, 1), (5, 2), (1, 3)]
```

Creative example: Sorting a list of student records based on their grades.

In [ ]:

```
# Creative Example: Sorting Student Records by Grades

class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

# Sample student records
students = [
    Student("Alice", 95),
    Student("Bob", 88),
    Student("Charlie", 92),
]

def sort_by_second(data):
    """
    Sorts a list of data elements based on a specified attribute.

    Args:
        data: A list of objects with a sortable attribute.

    Returns:
        A new list containing the elements sorted by the specified attribute.
    """
    return sorted(data, key=lambda x: x.grade)  # Access grade attribute

# Sort students by grade using the function
sorted_students = sort_by_second(students.copy())  # Sort a copy to avoid modify

# Print sorted student records with descriptive output
for student in sorted_students:
    print(f"{student.name}: {student.grade}")

# Output (example):
# Bob: 88
# Charlie: 92
# Alice: 95
```

```
Bob: 88
Charlie: 92
Alice: 95
```

9. **Python program using `map()` to convert Celsius to Fahrenheit:**

In [ ]:
```
def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32

celsius_temperatures = [0, 20, 30, 100]
fahrenheit_temperatures = list(map(celsius_to_fahrenheit, celsius))

print(fahrenheit_temperatures)  # Outputs [32.0, 68.0, 86.0, 212.0]
```

```
[32.0, 68.0, 86.0, 212.0]
```

Use case: Converting temperature data collected from different weather stations for reporting.

In [ ]:
```
def celsius_to_fahrenheit(celsius):
  """Converts a Celsius temperature to Fahrenheit.

  Args:
      celsius (float): The temperature in degrees Celsius.

  Returns:
      float: The temperature in degrees Fahrenheit.
  """

  # Conversion formula: Fahrenheit = (Celsius x 9/5) + 32
  return (celsius * 9/5) + 32

# Sample Celsius temperature data collected from weather stations
celsius_temperatures = [0, 20, 30, 100]

# Use map() to efficiently convert all Celsius temperatures
fahrenheit_temperatures = list(map(celsius_to_fahrenheit, celsius_temperatures))

# Print the converted temperatures with descriptive labels
print("Converted Temperatures (Fahrenheit):")
for i, temp in enumerate(fahrenheit_temperatures):
  print(f"Station {i+1}: {temp:.2f}°F")  # Format output to two decimal places
```

```
Converted Temperatures (Fahrenheit):
Station 1: 32.00°F
Station 2: 68.00°F
Station 3: 86.00°F
Station 4: 212.00°F
```

10. **Python program using `filter()` to remove all the vowels from a given string:**

In [ ]:
```
def remove_vowels(s):
    return ''.join(filter(lambda x: x.lower() not in 'aeiou', s))

# Example
result = remove_vowels("Hello World")

print(result)  # Outputs "Hll Wrld"
```

```
Hll Wrld
```

Creative example: Removing vowels from usernames to create unique and compact IDs.

In [ ]:
```python
def remove_vowels(s):
  """Removes vowels from a string and returns the consonant-only version.

  Args:
      s: The input string.

  Returns:
      The string with vowels removed.
  """

  consonants = ''.join(filter(lambda x: x.lower() not in 'aeiou', s))
  return consonants

def generate_compact_id(username):
  """Generates a unique and compact ID from a username by removing vowels.

  Args:
      username: The username to convert into a compact ID.

  Returns:
      A compact ID derived from the username.
  """

  compact_id = remove_vowels(username)[:8]  # Limit ID length to 8 characters
  return compact_id.lower()  # Ensure lowercase for consistency

# Example usage
usernames = ["StackOverflow", "CodingEnthusiast", "Pythonista"]
for username in usernames:
  compact_id = generate_compact_id(username)
  print(f"Username: {username}, Compact ID: {compact_id}")
```

```
Username: StackOverflow, Compact ID: stckvrfl
Username: CodingEnthusiast, Compact ID: cdngnths
Username: Pythonista, Compact ID: pythnst
```

11. **Python program for an accounting routine used in a book shop:**

In [ ]:
```python
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]
]

def calculate_order(order):
    """
    Calculate the total cost for a given order.
    If the total cost is less than $100, an additional fee of $10 is added.

    Parameters:
    order (list): A list containing the order details:
                  [order_number, book_title, quantity, price_per_item]

    Returns:
    tuple: A tuple containing the order number and the total cost.
    """
    order_num, _, quantity, price_per_item = order
```

```
    total_cost = quantity * price_per_item
    # Apply a small order handling fee if the total cost is below $100
    if total_cost < 100:
        total_cost += 10
    return (order_num, total_cost)

# Using map to apply the calculate_order function to each order in the orders li
order_totals = list(map(calculate_order, orders))

# Print the order totals
print("Order totals including handling fee for small orders:")
for order in order_totals:
    print(f"Order Number: {order[0]}, Total Cost: ${order[1]:.2f}")

# Output:
# Order Number: 34587, Total Cost: $163.80
# Order Number: 98762, Total Cost: $284.00
# Order Number: 77226, Total Cost: $108.85
# Order Number: 88112, Total Cost: $84.97
```

```
Order totals including handling fee for small orders:
Order Number: 34587, Total Cost: $163.80
Order Number: 98762, Total Cost: $284.00
Order Number: 77226, Total Cost: $108.85
Order Number: 88112, Total Cost: $84.97
```

Use case: Automating the calculation of order totals including handling of small orders with an additional fee.

Generate Summary Report Now, we will create a function to generate a summary report for the orders, including the total revenue and the number of orders processed.

In [ ]:

```
def generate_report(orders):
    """
    Generate a summary report for the list of orders.

    Parameters:
    orders (list): A list of orders where each order is a tuple (order_number, t

    Returns:
    None
    """
    total_revenue = sum(order[1] for order in orders)
    num_orders = len(orders)

    print("\nSummary Report")
    print("====================")
    print(f"Total Revenue: ${total_revenue:.2f}")
    print(f"Number of Orders: {num_orders}")

# Generate and print the summary report
generate_report(order_totals)
```

```
Summary Report
====================
Total Revenue: $641.62
Number of Orders: 4
```

End of Assignment 3!