THEORY QUESTION

QUESTION1: What is the difference between a function and a method in Python?

ANSWER1: The difference between a function and a method in Python, along with an example:

Function

- A function is a block of code that performs a specific task.
- It is defined using the def keyword.
- It can be called independently.

Method

- A method is a function that is associated with an object.
- It is defined within a class.
- It is called using the dot operator (.) on an object of the class.
  Example

```
[ ]
```

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


class Dog:
 def bark(self):
  print("Woof!")


greet("Alice")
```

```
my_dog = Dog()

my_dog.bark()
```

```
Hello, Alice!
Woof!
```

In this example:

- greet() is a function that takes a name as an argument and prints a greeting.
- bark() is a method that is associated with the Dog class. It prints "Woof!" when called on an object of the Dog class.
    - Key difference:
- The key difference between a function and a method is that a method is associated with an object, while a function is not. This means that a method can access and modify the data within the object it is associated with, while a function cannot.

QUESTION2: Explain the concept of function arguments and parameters in Python.

ANSWER2: Function arguments and parameters in Python with a single example:

```
[ ]
```

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

  print(f"{greeting}, {name}!")



greet("Gitanjali")

greet("Bob", greeting="Hi there")
```

```
Hello, Gitanjali!
Hi there, Bob!
```

---

1. def greet(name, greeting="Hello"):: This defines a function named greet. name and greeting are parameters. greeting has a default value of "Hello".
2. greet("Gitanjali"): We call greet with one argument, "Gitanjali". This is assigned to the name parameter. Since we don't provide a value for greeting, it uses the default "Hello".
3. greet("Bob", greeting="Hi there"): Here, we provide two arguments: "Bob" (assigned to name) and "Hi there" (assigned to greeting using a keyword argument). This overrides the default greeting.

QUESTION3: What are the different ways to define and call a function in Python?

ANSWER3: Defining a Function

```
[ ]
```

```
def greet(name):

  """This function greets the person passed in as a parameter."""

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

- def greet(name):: This line defines a function named greet that accepts one parameter called name.
- """This function greets the person passed in as a parameter.""": This is a docstring that describes what the function does.
- print(f"Hello, {name}!"): This line prints a greeting message to the console, using an f-string to insert the value of the name parameter into the message. Calling a Function

---

```
[ ]
```

```
greet("bebo")
```

```
Hello, bebo!
```

- greet("bebo"): This line calls the greet function and passes the string "bebo" as an argument. The value "bebo" is assigned to the name parameter inside the function.

The greet function takes the name "bebo" as input, inserts it into the greeting message, and then prints the message to the console. This is the expected output when the function is called with the argument "bebo".

QUESTION4: What is the purpose of the `return` statement in a Python function?

ANSWER4: Purpose of the return statement in Python Functions

The return statement is used to terminate the execution of a function and optionally return a value to the caller.

Key Functions:

1. Termination: When the return statement is encountered inside a function, the function immediately stops executing, and control returns to the point where the function was called. Any code after the return statement within the function is not executed.
2. Returning a Value: The return statement is often used to send a value back to the caller of the function. This value can be any data type in Python, such as an integer, string, list, tuple, or even another function. If no value is specified after the return keyword, the function returns None implicitly.

Example:

```
[ ]
```

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

rectangle_area = calculate_area(5, 4)
print(f"The area of the rectangle is: {rectangle_area}")
```

```
The area of the rectangle is: 20
```

The return statement is essential for functions to produce and return results that can be used by other parts of a program. It enables functions to perform calculations, process data, and provide meaningful outputs to the caller, making them versatile and powerful building blocks in Python programming.

QUESTION5: What are iterators in Python and how do they differ from iterables?

ANSWER5:

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.
- Examples: lists, tuples, strings, dictionaries, sets, files.
- They have an **iter**() method which returns an iterator.

Iterators

- An iterator is an object representing a stream of data.
- It implements two methods:
  - **iter**(): Returns the iterator object itself.
  - **next**(): Returns the next item in the stream. Raises a StopIteration exception when there are no more items.
- You can get an iterator from an iterable using the iter() function.

Example

```
[ ]
```

```
my_list = [2, 3, 5]


my_iterator = iter(my_list)
```

```
print(next(my_iterator))

print(next(my_iterator))

print(next(my_iterator))
```

```
2
3
5
```

In the example:

1. my_list is an iterable (a list).
2. my_iterator is an iterator obtained using iter(my_list).
3. We use next(my_iterator) to get the next item from the iterator.

QUESTION6: Explain the concept of generators in Python and how they are defined.

ANSWER6:

Concept of Generators

- Generators are special functions that produce a sequence of values using the yield keyword instead of return.
- They don't compute all values at once, but generate them on demand (lazy evaluation).
- This makes them memory-efficient for large datasets, as they only store one value at a time.
- They are iterable, meaning you can loop through them using a for loop.

Defining Generators

1. Function with yield: Create a function that includes the yield keyword.
2. Calling the Generator: When called, the function returns a generator object.
3. Iteration: Use next() or a for loop to retrieve values from the generator.

Example

```
[ ]
```

```
def countdown(n):
 while n > 0:
  yield n
  n -= 1
counter = countdown(8)


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

```
8
7
6
5
4
3
2
1
```

---

1. countdown(n) is a generator function.
2. yield n yields the current value of n and pauses execution.
3. The loop iterates, resuming from where it left off and yielding the next value until n reaches 0.

QUESTION7: What are the advantages of using generators over regular functions?

ANSWER7: Advantages of Generators over Regular Functions:

1. Memory Efficiency: Generators are memory-efficient because they generate values on demand instead of storing them all in memory at once. This is especially beneficial when dealing with large datasets or infinite sequences, as it avoids loading the entire dataset into memory.
2. Improved Performance: Generators can improve performance due to lazy evaluation. Values are generated only when needed, leading to faster execution, especially in cases where not all values may be required.

3. Readability: Generators can make code more concise and readable when dealing with sequences or iterations. The yield keyword simplifies the process of generating values step-by-step.
4. Representing Infinite Streams: Generators can represent infinite streams of data, which is impossible with regular functions that would need to store an infinite amount of data.
5. Pipelining Generators: Generators can be chained together, allowing you to process data in a pipeline fashion, where the output of one generator becomes the input of another, further enhancing efficiency.

Example

Let's consider an example to illustrate the memory efficiency of generators:

Regular Function:

```
[ ]
```

```
def squares(n):
    result = []
    for i in range(n):
        result.append(i * i)
    return result


squares_list = squares(10)
print(squares_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Generator:

```
[ ]
```

```
def squares_generator(n):
```

```
    for i in range(n):
        yield i * i

squares_gen = squares_generator(10)
for square in squares_gen:
    print(square)
```

```
0
1
4
9
16
25
36
49
64
81
```

In this example, the squares function stores all squares in a list, consuming memory for all values. The squares_generator, on the other hand, generates each square one at a time when needed, using less memory.

QUESTION8: What is a lambda function in Python and when is it typically used?

ANSWER8: A lambda function is a small, anonymous function defined using the lambda keyword. It can have any number of arguments but can only have one expression. Lambda functions are often used for short, simple operations where defining a full function using def would be overkill.

- When is it typically used?

Lambda functions are typically used in situations where you need a simple function for a short period of time and don't want to define a formal function using def. They

are commonly used as arguments to higher-order functions like map, filter, and reduce.

Example:

```
[ ]
```

```
square = lambda x: x**2


result = square(8)

print(result)
```

```
64
```

QUESTION9: Explain the purpose and usage of the `map()` function in Python.

ANSWER9:

Purpose:

The map() function applies a given function to each item of an iterable (like a list, tuple, etc.) and returns an iterator containing the results. It essentially "maps" the function across the iterable's elements.

Usage:

- function: The function to be applied to each element.
- iterable: The iterable containing the elements to be processed.
- ...: Optional - Additional iterables if the function takes multiple arguments.

Example:

```
[ ]
```

```
numbers = [1, 3, 4, 6, 8]

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

print(squared_numbers)
```

```
[1, 9, 16, 36, 64]
```

QUESTION10: What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

ANSWER10:

1. map()

- Purpose: Applies a function to each item in an iterable and returns an iterator with the results.
- Focus: Transformation of elements. Example:

```
[ ]
```

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

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

print(squared_numbers)
```

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

---

2. filter()

- Purpose: Filters elements from an iterable based on a condition specified by a function.
- Focus: Selection of elements.
- Requires the functools library in Python 3+ Example:

```
[ ]
```

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

```
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even_numbers)
```

```
[2, 4]
```

---

3. reduce()

- Purpose: Applies a function cumulatively to the items of an iterable, reducing them to a single value.
- Focus: Aggregation of elements.
- Requires the functools library in Python 3+ Example:

```
[ ]
```

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

product = reduce(lambda x, y: x * y, numbers)

print(product)
```

```
120
```

PRACTICAL QUESTIONS

QUESTION1: Write a Python function that takes a list of numbers as input and returns the sum of all even numbers in the list.

ANSWER1:

```
[ ]
```

```
def sum_even_numbers(numbers):

  even_sum = 0

  for number in numbers:
```

```
        if number % 2 == 0:
            even_sum += number
    return even_sum


num_list = [10, 15, 20, 25, 30]
result = sum_even_numbers(num_list)
print("Sum of even numbers:", result)
```

```
Sum of even numbers: 60
```

QUESTION2: Create a Python function that accepts a string and returns the reverse of that string.

ANSWER2:

```
[ ]
```

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


string = "hello"
reversed_string = reverse_string(string)
print(reversed_string)
```

```
olleh
```

QUESTION3: Implement a Python function that takes a list of integers and returns a new list containing the squares of each number.

ANSWER3:

```
[ ]
```

```
def square_numbers(numbers):
  squared_numbers = []
  for number in numbers:
    squared_numbers.append(number**2)
  return squared_numbers


numbers = [2, 4, 6, 5, 9]
squared_list = square_numbers(numbers)
print(squared_list)
```

```
[4, 16, 36, 25, 81]
```

QUESTION4: Write a Python function that checks if a given number is prime or not from 1 to 200.

ANSWER4:

```
[ ]
```

```
def is_prime(number):
  if number < 2:
    return False
```

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

# Checking prime numbers from 1 to 200
for num in range(1, 201):
    if is_prime(num):
        print(num, end=" ")
```

```
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
```

QUESTION5: Create an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms.

ANSWER5:

```
[ ]
```

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

```
    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.num_terms:
            raise StopIteration
        if self.count == 0:
            self.count += 1
            return self.a
        elif self.count == 1:
            self.count += 1
            return self.b
        else:
            next_value = self.a + self.b
            self.a, self.b = self.b, next_value
            self.count += 1
            return next_value

fib = FibonacciIterator(10)
for number in fib:
    print(number, end=" ")
```

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

QUESTION6: Write a generator function in Python that yields the powers of 2 up to a given exponent. ANSWER6:

```
[ ]
```

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

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

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

QUESTION7: Implement a generator function that reads a file line by line and yields each line as a string.

ANSWER7:

```
[ ]
```

```
# Generator function to read file line by line
def read_file_line_by_line(file_path):
    with open(file_path, 'r') as file:
        for line in file:
```

```
        yield line.strip()  # Remove leading/trailing whitespace


# Using the generator
file_name = '/Untitled document.txt'
# Make sure this file exists
for line in read_file_line_by_line(file_name):
    print(line)
```

```
Hi Hello
```

QUESTION8: Use a lambda function in Python to sort a list of tuples based on the second element of each tuple.

ANSWER8:

```
[ ]
```

```
# List of tuples
my_list = [(1, 5), (2, 3), (4, 1), (3, 7)]


# Sort by the second element using lambda
sorted_list = sorted(my_list, key=lambda x: x[1])


print(sorted_list)
```

```
[(4, 1), (2, 3), (1, 5), (3, 7)]
```

QUESTION9: Write a Python program that uses `map()` to convert a list of temperatures from Celsius to Fahrenheit.

ANSWER9:

```
[ ]
```

```
# List of temperatures in Celsius

celsius = [0, 20, 30, 40, 100]



# Function to convert Celsius to Fahrenheit

def celsius_to_fahrenheit(temp):

   return (temp * 9/5) + 32



# Use map() to apply the conversion

fahrenheit = list(map(celsius_to_fahrenheit, celsius))



print("Temperatures in Fahrenheit:", fahrenheit)
```

```
Temperatures in Fahrenheit: [32.0, 68.0, 86.0, 104.0, 212.0]
```

QUESTION10: Create a Python program that uses `filter()` to remove all the vowels from a given string.

ANSWER10:

```
[ ]
```

```
def remove_vowels(string):
  vowels = "aeiouAEIOU"

  filtered_string = filter(lambda char: char not in vowels, string)

  result = "".join(filtered_string)
  return result

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

```
Hll, Wrld!
```

QUESTION11: Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:

ANSWER11:

```
[ ]
```

```
data = [
  ("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 Python, Bernd Klein", 2, 24.00),
]

def calculate_total_price(item):
    order_number, title_author, quantity, price_per_item = item
    return quantity * price_per_item

total_prices = list(map(calculate_total_price, data))

# Calculate the total value of the order
total_order_value = sum(total_prices)

print(f"Total prices for each item: {total_prices}")
print(f"Total value of the order: {total_order_value:.2f} €")

if total_order_value > 100:
    increased_value = total_order_value * 1.15
    print(f"Since the total order value exceeds 100 €, the increased value is:
{increased_value:.2f} €")
```

```
Total prices for each item: [163.8, 284.0, 98.85000000000001, 48.0]
Total value of the order: 594.65 €
Since the total order value exceeds 100 €, the increased value is:
683.85 €
```