Q1. Which keyword is used to create a function? Create a function to return a list of odd numbers in the

range of 1 to 25.

to create a function in most programming languages is typically "def" (short for "define"). However, it's important to note that different programming languages may have variations in syntax and keywords.

Here's an example of a Python function that returns a list of odd numbers in the range of 1 to 25:

def get\_odd\_numbers():

odd\_numbers = []

for number in range(1, 26):

if number % 2 != 0:

odd\_numbers.append(number)

return odd\_numbers

# Call the function and print the result

print(get\_odd\_numbers())

In this example, the function is named "get\_odd\_numbers" and it initializes an empty list called "odd\_numbers". It then iterates through the range of numbers from 1 to 25 and checks if each number is odd by using the modulo operator (%). If the number is odd, it is appended to the "odd\_numbers" list. Finally, the function returns the list of odd numbers, and we print the result by calling the function and using the "print" statement.

Q2. Why \*args and \*\*kwargs is used in some functions? Create a function each for \*args and \*\*kwargs

to demonstrate their use.

The ‘**\*args’** and ‘**\*\*kwargs’** are special syntax in Python that allow a function to accept a variable number of arguments.

The ‘**\*args’** parameter in a function definition allows you to pass any number of positional arguments to the function. The name "args" is a convention, but you can actually use any name preceded by a single asterisk. Inside the function, ‘**args’** becomes a tuple containing all the positional arguments passed to the function.

Here's an example of a function that accepts ‘**\*args’**:

def concatenate\_strings(\*args):

result = ""

for arg in args:

result += arg

return result

In this function, you can pass any number of strings as arguments, and the function will concatenate them together and return the result. For example:

print(concatenate\_strings("Hello", " ", "world", "!"))

# Output: "Hello world!"

The ‘**\*\*kwargs’** parameter allows you to pass any number of keyword arguments to a function. Similar to ‘**\*args’**, the name "kwargs" is a convention, but you can use any name preceded by a double asterisk. Inside the function, ‘**kwargs’** becomes a dictionary containing the keyword arguments passed to the function, where the keys are the argument names and the values are the corresponding values.

Here's an example of a function that accepts ‘**\*\*kwargs’**:

def display\_person\_info(\*\*kwargs):

for key, value in kwargs.items():

print(key + ": " + value)

In this function, you can pass any number of keyword arguments representing person information (e.g., name, age, location), and the function will display them. For example:

display\_person\_info(name="John", age="30", location="New York")

# Output:

# name: John

# age: 30

# location: New York

Note that you can also use a combination of ‘**\*args’** and ‘**\*\*kwargs’** in a function definition if you want to accept both positional and keyword arguments.

Q3. What is an iterator in python? Name the method used to initialise the iterator object and the method

used for iteration. Use these methods to print the first five elements of the given list [2, 4, 6, 8, 10, 12, 14,

16, 18, 20].

In Python, an iterator is an object that allows you to iterate over a collection of elements, such as a list, tuple, or dictionary, one element at a time. It provides a way to access the elements sequentially without exposing the underlying implementation.

To initialize an iterator object in Python, you can use the ‘**iter()’** function. It takes an iterable object as an argument and returns an iterator object. For example, to initialize an iterator for a list, you would use the’**iter()’** function as follows:

my\_list = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

my\_iterator = iter(my\_list)

Once you have an iterator object, you can use the **next()** function to retrieve the next element in the iteration sequence. It advances the iterator to the next element and returns it. You can continue calling **next()** to access subsequent elements. However, if there are no more elements in the sequence, it raises the **StopIteration** exception. To iterate over all the elements in an iterator, you can use a loop.

To print the first five elements of the given list **[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]** using an iterator, you can follow this code:

my\_list = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

my\_iterator = iter(my\_list)

for \_ in range(5):

element = next(my\_iterator)

print(element)

Output:

2

4

6

8

10

In this example, we initialize an iterator object ‘**my\_iterator’** using the ‘**iter()’** function on the ‘**my\_list’** list. Then, we use a loop to iterate five times, calling ‘**next(my\_iterator)’** in each iteration to retrieve the next element and print it.

Q4. What is a generator function in python? Why yield keyword is used? Give an example of a generator

function.

In Python, a generator function is a special type of function that returns an iterator, which can be iterated over to retrieve values one at a time. Unlike regular functions that return a value and then terminate, generator functions can pause and resume their execution, allowing them to generate a sequence of values dynamically. They are useful when dealing with large data sets or when you don't want to generate all the values at once.

The ‘**yield’** keyword is used in generator functions to define points at which the function should pause and return a value to the caller. When a generator function is called, it returns a generator object, which can be iterated over using a’**for’** loop or by using the ‘**next()’** function. Each time the ‘**yield’** keyword is encountered, the function's state is saved, and the yielded value is returned. When the generator is iterated over again, it resumes execution from where it left off, allowing it to generate the next value in the sequence.

Here's an example of a generator function that generates a sequence of even numbers up to a given limit:

def even\_numbers(limit):

n = 0

while n <= limit:

yield n

n += 2

# Using the generator function

even\_gen = even\_numbers(10)

# Iterating over the generator using a for loop

for num in even\_gen:

print(num)

In this example, the ‘**even\_numbers’** function is a generator function that generates even numbers. Each time the ‘**yield’** statement is encountered, the current value of ‘**n’** is returned, and the function's state is saved. The ‘**for’** loop iterates over the generator, printing each even number up to the limit of 10. The generator function only generates the next number in the sequence when requested, which makes it memory-efficient and suitable for generating large sequences.

Q5. Create a generator function for prime numbers less than 1000. Use the next() method to print the

first 20 prime numbers.

def prime\_generator():

yield 2 # 2 is the first prime number

primes = [2]

num = 3

while num < 1000:

is\_prime = True

for prime in primes:

if prime \* prime > num:

break

if num % prime == 0:

is\_prime = False

break

if is\_prime:

yield num

primes.append(num)

num += 2 # Only check odd numbers for efficiency

# Create the prime number generator

prime\_gen = prime\_generator()

# Print the first 20 prime numbers

for \_ in range(20):

prime = next(prime\_gen)

print(prime)

When you run this code, it will generate and print the first 20 prime numbers less than 1000:

2

3

5

7

11

13

17

19

23

29

31

37

41

43

47

53

59

61

67

71

The ‘**prime\_generator’** function is a generator that yields prime numbers one by one using the Sieve of Eratosthenes algorithm. The ‘**next()’** method is used to obtain the next prime number from the generator.

Q6. Write a python program to print the first 10 Fibonacci numbers using a while loop.

# Initialize the first two numbers of the Fibonacci sequence

num1, num2 = 0, 1

# Initialize a counter for the number of Fibonacci numbers printed

count = 0

# Print the first 10 Fibonacci numbers using a while loop

while count < 10:

print(num1)

temp = num1 + num2

num1 = num2

num2 = temp

count += 1

When you run this program, it will output the following sequence of numbers:

0

1

1

2

3

5

8

13

21

34

Each number is obtained by adding the previous two numbers in the sequence. The loop continues until 10 Fibonacci numbers are printed.

Q7. Write a List Comprehension to iterate through the given string: ‘pwskills’.

Expected output: ['p', 'w', 's', 'k', 'i', 'l', 'l', 's']

string = 'pwskills'

output = [char for char in string]

print(output)

Output:

['p', 'w', 's', 'k', 'i', 'l', 'l', 's']

In the above code, ‘**char’** represents each character in the string, and it is added to the list comprehension as ‘**char’** itself. The resulting list is stored in the variable ‘**output** ‘and then printed.

Q8. Write a python program to check whether a given number is Palindrome or not using a while loop.

def is\_palindrome(number):

original\_number = number

reversed\_number = 0

while number > 0:

# Extract the last digit

digit = number % 10

# Append the digit to the reversed number

reversed\_number = reversed\_number \* 10 + digit

# Remove the last digit from the original number

number //= 10

# Check if the original number and the reversed number are equal

if original\_number == reversed\_number:

return True

else:

return False

# Test the function

num = int(input("Enter a number: "))

if is\_palindrome(num):

print(num, "is a palindrome")

else:

print(num, "is not a palindrome")

In this program, the ‘**is\_palindrome’** function takes a number as input and returns ‘**True’** if it is a palindrome and ‘**False’** otherwise. The function uses a while loop to reverse the number and checks if the reversed number is equal to the original number.

You can run the program and enter a number to check whether it is a palindrome or not.

Q9. Write a code to print odd numbers from 1 to 100 using list comprehension.

Note: Use a list comprehension to create a list from 1 to 100 and use another List comprehension to filter

out odd numbers.

# Creating a list of numbers from 1 to 100

numbers = [x for x in range(1, 101)]

# Filtering out odd numbers using another list comprehension

odd\_numbers = [x for x in numbers if x % 2 != 0]

# Printing the odd numbers

print(odd\_numbers)

When you run this code, it will generate a list of odd numbers from 1 to 100 using list comprehension and then print the resulting list.