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.

Ans. In Python, the keyword used to create a function is `def`. Here's an example of a function that returns a list of odd numbers in the range of 1 to 25:

```
def odd_numbers():
   odd_list = [num for num in range(1, 26) if num % 2 != 0]
   return odd_list

# Call the function
result = odd_numbers()
print(result)
```

Q2. Why *args and **kwargs is used in some functions? Create a function each for *args and **kwargs to demonstrate their use.

Ans. In Python, *args and **kwargs are used to handle variable numbers of arguments in a function.

1. *args (Arbitrary Arguments):
The *args syntax allows a function to accept a variable number of positional arguments. It collects these arguments into a tuple. Here's an example:

```
def sum_values(*args):
   total = 0
   for num in args:
      total += num
   return total

result = sum_values(1, 2, 3, 4, 5)
print(result)
```

In this example, the `sum_values` function can accept any number of arguments, and it calculates the sum of all the provided values.

1. **kwargs (Arbitrary Keyword Arguments):

The `**kwargs` syntax allows a function to accept a variable number of keyword arguments. It collects these arguments into a dictionary. Here's an example:

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="John", age=25, city="Exampleville")
```

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

print_info(name="John", age=25, city="Exampleville")

In this example, the print_info function can accept any number of keyword arguments, and it prints out the key-value pairs.

These features are especially useful when you want to create more flexible and generic functions that can handle different input configurations.

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].

Ans. In Python, an iterator is an object that represents a stream of data and can be iterated (looped) over. It implements two methods: **__iter__() and __next__().**

__iter__() method initializes the iterator object and returns itself.
__next__() method retrieves the next element from the iterator.
Here's an example of creating and using an iterator to print the first five elements of the given list:

```
# Create an iterator for the list [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
my_list = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
my_iterator = iter(my_list)

# Use the iterator to print the first five elements
for _ in range(5):
   element = next(my_iterator)
   print(element)
```

In this example, **iter(my_list)** initializes the iterator object for the list, and **next(my_iterator**) retrieves the next element in each iteration of the loop. The loop runs five times, printing the first five elements of the list.

Keep in mind that when you reach the end of the iterator, further calls to **next()** will raise a **StopIteration** exception. It's a good practice to handle this exception if you're unsure about the length of the iterable.

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

Ans. A generator function in Python is a special type of function that allows you to iterate over a potentially large sequence of data without actually storing the entire sequence in memory. Generator functions use the **yield** keyword to produce a series of values one at a time during each iteration.

The **yield** keyword is used in a generator function to temporarily suspend the function's state, allowing it to be resumed later. When the generator function is called, it doesn't execute the whole function at once. Instead, it yields a value and temporarily halts its execution until the next value is requested.

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

```
def generate_squares(limit):
    n = 1
    while n <= limit:
        yield n ** 2
        n += 1

# Using the generator function to print squared numbers up to 10
squares_generator = generate_squares(10)

for square in squares_generator:
    print(square)
```

In this example, **generate_squares** is a generator function that yields the square of numbers from 1 to the specified **limit**. When you iterate over the generator using a **for** loop, the function is executed until it reaches the **yield** statement, produces a value, and then pauses. The state of the function is preserved, and it resumes from where it left off when the next value is requested.

Using generators is memory-efficient for large datasets, as they don't require storing the entire sequence in memory at once. Instead, values are generated on-the-fly during each iteration.

Q5. Create a generator function for prime numbers less than 1000. Use the next() method to print the first 20 prime numbers.

Ans. Certainly! Here's an example of a generator function that yields prime numbers less than 1000, and then it uses the **next()** method to print the first 20 prime numbers:

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

def generate_primes():
   num = 2
   count = 0
   while count < 20:
      if is_prime(num):
         yield num
         count += 1
      num += 1

# Using the generator function to print the first 20 prime numbers
prime_generator = generate_primes()

for _ in range(20):
   prime = next(prime_generator)
   print(prime)
```

In this example, the *is_prime* function checks whether a given number is prime. The *generate_primes* generator function yields prime numbers until it has generated the first 20 prime numbers. The *next()* method is used in a loop to obtain and print each prime number from the generator. The loop runs 20 times, printing the first 20 prime numbers less than 1000.

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

Ans. The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones, usually starting with 0 and 1. Here's a Python program using a while loop to print the first 10 Fibonacci numbers:

```
# Function to print the first n Fibonacci numbers
def fibonacci(n):
    fib_sequence = [0, 1]

    while len(fib_sequence) < n:
        next_number = fib_sequence[-1] + fib_sequence[-2]
        fib_sequence.append(next_number)

    return fib_sequence

# Print the first 10 Fibonacci numbers
first_10_fibonacci = fibonacci(10)
print(first_10_fibonacci)
```

In this program, the `fibonacci` function uses a while loop to generate the Fibonacci sequence up to the specified count (`n`). It starts with the initial values `[0, 1]` and continues to calculate the next Fibonacci numbers until the sequence reaches the desired length. The result is then printed. In this case, it prints the first 10 Fibonacci

Q7. Write a List Comprehension to iterate through the given string: 'pwskills'.
Expected output: ['p', 'w', 's', 'k', 'i', 'l', 'l', 's']

Ans. You can use a list comprehension to iterate through the given string 'pwskills' and create a list of its individual characters. Here's how you can achieve this:

```
input_string = 'pwskills'
result_list = [char for char in input_string]
print(result_list)
```

This list comprehension iterates through each character (`char`) in the string 'pwskills' and creates a new list with those characters. The output will be:

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

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

Ans. A palindrome number is one that reads the same backward as forward. Here's a Python program that checks whether a given number is a palindrome or not using a while loop:

```
def is_palindrome(number):
    original_number = number
    reversed_number = 0

    while number > 0:
        digit = number % 10
        reversed_number = reversed_number * 10 + digit
        number = number // 10

    return original_number == reversed_number

# Input number to check for palindrome
input_number = int(input("Enter a number: "))

# Check if the input number is a palindrome
if is_palindrome(input_number):
    print(f"{input_number} is a palindrome.")
else:
    print(f"{input_number} is not a palindrome.")
```

In this program, the `is_palindrome` function takes a number as input and uses a while loop to reverse the digits of the number. It then compares the original number with its reversed version to determine if it's a palindrome. The result is printed based on the check.

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.

Ans. You can use list comprehensions to achieve this. Here's a code snippet that prints odd numbers from 1 to 100 using list comprehensions:

```
# Using list comprehension to create a list from 1 to 100
numbers = [num for num in range(1, 101)]

# Using another list comprehension to filter out odd numbers
odd_numbers = [num for num in numbers if num % 2 != 0]

# Printing the result
print(odd_numbers)
```

In this code:

1.The first list comprehension creates a list numbers containing numbers from 1 to 100.
2.The second list comprehension (odd_numbers) filters out odd numbers from the numbers list using the condition num % 2 != 0.
3.The result is printed, which is a list containing all odd numbers from 1 to 100.