1. The keyword used to create a function in Python is "def". Here is an example of how to create a function to return a list of odd numbers in the range of 1 to 25:

def get\_odd\_numbers():

odd\_numbers = []

for i in range(1, 26):

if i % 2 == 1:

odd\_numbers.append(i)

return odd\_numbers

In this function, we use the "def" keyword to define the function name "get\_odd\_numbers". We then create an empty list called "odd\_numbers". We loop through the range of numbers from 1 to 26 and use an if statement to check if each number is odd. If it is, we append it to the "odd\_numbers" list. Finally, we return the list of odd numbers.

1. \*args and \*\*kwargs are special syntax in Python for passing a variable number of arguments to a function.

\*args is used to pass a variable number of non-keyword arguments to a function. It allows us to pass any number of arguments to a function, separated by commas. These arguments are then passed to the function as a tuple.

\*\*kwargs is used to pass a variable number of keyword arguments to a function. It allows us to pass any number of arguments to a function, but these arguments must be passed as key value pairs. These key-value pairs are then passed to the function as a dictionary.

Here are example functions to demonstrate the use of \*args and \*\*kwargs:

def sum\_numbers(\*args):

sum = 0

for num in args:

sum += num

return sum

def print\_values(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

print(sum\_numbers(1, 2, 3, 4, 5)) # Output: 15

print\_values(name="Alice", age=25, country="USA") # Output: name: Alice, age: 25, country: USA

In the first function, we use \*args to pass a variable number of non-keyword arguments. We then loop through the arguments and add them together to get the sum.

In the second function, we use \*\*kwargs to pass a variable number of keyword arguments. We then loop through the key-value pairs in the dictionary and print them out.

1. \*args and \*\*kwargs are special syntax in Python for passing a variable number of arguments to a function.

\*args is used to pass a variable number of non-keyword arguments to a function. It allows us to pass any number of arguments to a function, separated by commas. These arguments are then passed to the function as a tuple.

\*\*kwargs is used to pass a variable number of keyword arguments to a function. It allows us to pass any number of arguments to a function, but these arguments must be passed as key value pairs. These key-value pairs are then passed to the function as a dictionary.

Here are example functions to demonstrate the use of \*args and \*\*kwargs:

def sum\_numbers(\*args):

sum = 0

for num in args:

sum += num

return sum

def print\_values(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

print(sum\_numbers(1, 2, 3, 4, 5)) # Output: 15

print\_values(name="Alice", age=25, country="USA") # Output: name: Alice, age: 25, country: USA

In the first function, we use \*args to pass a variable number of non-keyword arguments. We then loop through the arguments and add them together to get the sum.

In the second function, we use \*\*kwargs to pass a variable number of keyword arguments. We then loop through the key-value pairs in the dictionary and print them out.

1. A generator function in Python is a special type of function that returns an iterator, which can be used to iterate over a sequence of values. It allows you to generate a sequence of values on the fly, without having to create and store the entire sequence in memory at once.

The `yield` keyword is used in a generator function to return a value and suspend the function’s execution. When the generator’s `next()` method is called again, the function resumes execution from where it left off, and continues until it hits another `yield` statement or reaches the end of the function.

Here is an example of a generator function that generates a sequence of even numbers:

def generate\_even\_numbers(n):

for i in range(n):

if i % 2 == 0:

yield i

In this generator function, we use a `for` loop to iterate over the range of numbers from 0 to `n`. We use an `if` statement to check if each number is even, and if it is, we use the `yield` keyword to return the number and suspend the function’s execution.

We then create a generator object called `even\_numbers` by calling the `generate\_even\_numbers()` function with an argument of 10. We use a `for` loop to iterate over the generator object and print out each even number.

Note that the `generate\_even\_numbers()` function does not create and store the entire sequence of even numbers in memory at once. Instead, it generates each number on the fly as it is needed, which can be more memory-efficient for large sequences

5) Here is an example of a generator function for prime numbers less than 1000, and how to use the `next()` method to print the first 20 prime numbers:

def generate\_primes():

# Generate prime numbers less than 1000

for num in range(2, 1000):

if all(num % i != 0 for i in range(2, int(num\*\*0.5)+1)):

yield num

# Example usage:

prime\_generator = generate\_primes()

for i in range(20):

print(next(prime\_generator))

In this code, we define a generator function called `generate\_primes()` that generates prime numbers less than 1000. We use a `for` loop to iterate over the range of numbers from 2 to 1000. For each number, we use a generator expression with the `all()` function to check whether it is prime. If it is, we use the `yield` keyword to return the number and suspend the function’s execution.

We then create a generator object called `prime\_generator` by calling the `generate\_primes()` function. We use a `for` loop and the `next()` function to iterate over the first 20 prime numbers generated by the generator, printing each one out as we go.

Note that because we are using a generator function, the prime numbers are generated on the fly as we iterate over them, rather than being generated all at once and stored in memory. This can be more memory-efficient for large sequences.