Ans 1 The keyword used to create a function is def. Here's how we can create a function to return a list of odd numbers in the range of 1 to 25 using Python:

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

# Call the function and store the result
result = get_odd_numbers()

print(result)  # This will print the list of odd numbers
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25]
```

ANS 2 *args and **kwargs are used in Python functions to handle variable-length argument lists. They allow us to pass a varying number of arguments to a function. Here's how they work: *args: The *args syntax allows a function to accept a variable number of positional arguments. It collects these arguments into a tuple within the function.

In [2]:
```
def print_arguments(*args):
    for arg in args:
        print(arg)

print_arguments('Hello', 'World', 123, [1, 2, 3])
```

```
Hello
World
123
[1, 2, 3]
```

In this example, the print_arguments function takes any number of arguments and prints each one. The output will be

Hello World 123 [1, 2, 3]

**kwargs: The **kwargs syntax allows a function to accept a variable number of keyword arguments. It collects these arguments into a dictionary within the function

In [7]:
```
def print_keyword_arguments(**kwargs):
    for key, value in kwargs.items():
        print(key, ":", value)

print_keyword_arguments(name='Alice', age=30, city='Wonderland')
```

```
name : Alice
age : 30
city : Wonderland
```

In this example, the print_keyword_arguments function takes any number of keyword arguments and prints each key-value pair. The output will be

we can use both *args and **kwargs in the same function definition. Here's an example that demonstrates their combined use

In [9]:
```
def combined_example(arg1, *args, kwarg1=None, **kwargs):
    print("arg1:", arg1)
    print("args:", args)
    print("kwarg1:", kwarg1)
    print("kwargs:", kwargs)

combined_example('first', 'second', 'third', kwarg1='keyword', key1='value1', key2=
```

```
arg1: first
args: ('second', 'third')
kwarg1: keyword
kwargs: {'key1': 'value1', 'key2': 'value2'}
```

Ans 3 In Python, an iterator is an object that allows us to traverse through a sequence of elements, such as a list, tuple, or other iterable data structures. Iterators provide a way to access the elements of a collection one by one, without the need to access the entire collection at once. The two main methods associated with iterators are: iter(): This method initializes an iterator object from an iterable. next(): This method is used to iterate through the elements of the iterator. It returns the next element in the sequence and advances the iterator. When there are no more elements to iterate through, it raises a StopIteration exception. Now, let's use these methods to print the first five elements of the given list [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

In [10]:
```python
# Given list
my_list = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

# Initialize an iterator object
my_iterator = iter(my_list)

# Iterate and print the first five elements
for _ in range(5):
    try:
        element = next(my_iterator)
        print(element)
    except StopIteration:
        break
```

```
2
4
6
8
10
```

Ans 4 A generator function in Python is a special type of function that returns an iterator called a generator. The yield keyword is used in a generator function to specify the values that the generator will produce. When a generator function is called, it doesn't execute the entire function body immediately. Instead, it starts executing until it encounters a yield statement. At that point, it yields the value specified by the yield keyword to the caller and temporarily pauses the function's execution state. When the generator is iterated again (using the next() function or a loop), the function resumes execution from where it was paused and continues until the next yield statement or until the function completes. Here's an example of a generator function that generates a sequence of squares:

In [11]:
```python
def square_generator(n):
    for i in range(n):
        yield i ** 2

# Create a generator object
squares = square_generator(5)

# Iterate through the generator
for square in squares:
    print(square)
```

```
0
1
4
9
16
```

Ans 5

In [13]:
```python
def is_prime(num):
    if num <= 1:
        return False
```

```
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True

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

# Create a generator object for prime numbers
prime_gen = prime_generator(20)

# Print the first 20 prime numbers using next()
for _ in range(20):
    prime = next(prime_gen)
    print(prime)
```

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
```

Ans 6

In [14]:

```
def fibonacci(n):
    fib_sequence = [0, 1]
    while len(fib_sequence) < n:
        next_fib = fib_sequence[-1] + fib_sequence[-2]
        fib_sequence.append(next_fib)
    return fib_sequence

# Get the first 10 Fibonacci numbers
first_10_fibonacci = fibonacci(10)

# Print the first 10 Fibonacci numbers
```

```
for num in first_10_fibonacci:
    print(num)
```

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

Ans 7

In [15]:
```
input_string = 'pwskills'

output_list = [char for char in input_string if char in 'wskil']
print(output_list)
```

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

Ans 8

In [ ]:
```
def is_palindrome(number):
    original_number = number
    reversed_number = 0
    while number > 0:
        digit = number % 10
        reversed_number = reversed_number * 10 + digit
        number //= 10
    return original_number == reversed_number

# Get input from the user
num = int(input("Enter a number: "))

if is_palindrome(num):
    print(f"{num} is a palindrome.")
else:
    print(f"{num} is not a palindrome.")
```

Ans 9

In [1]:
```
odd_numbers = [num for num in range(1, 101) if num % 2 != 0]

print(odd_numbers)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 4
3, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83,
85, 87, 89, 91, 93, 95, 97, 99]
```

In [ ]:
```

```