Answer 1

i - def keyword is used to create a function

ans1 - Part-ii

```
In [6]: def odd_numbers_in_range(start, end):
            odd_numbers = []
            for number in range(start, end+1 ):
                if number % 2 != 0:  # Check if the number is odd
                    odd_numbers.append(number)
            return odd_numbers
```

```
In [7]: odd_numbers_in_range(1, 25)
```

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

Answer 2 part i

*args is used to pass a variable number of non-keyword (positional) arguments to a function, It allows you to call a function with any number of arguments.

*kwargs is used to pass a variable number of keyword arguments (key-value pairs) to a function. It allows you to call a function with any number of keyword arguments.

Answer2 part ii

```
In [11]: def total(*args):
             return args
```

```
In [12]: total(1,2,5,36,4,7)
```

```
Out[12]: (1, 2, 5, 36, 4, 7)
```

```
In [14]: def key_word(**kwargs):
             return kwargs
```

```
In [16]: key_word(a = 25, b = 27, c=30)
```

```
Out[16]: {'a': 25, 'b': 27, 'c': 30}
```

Answer 3 part i

An iterator in Python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets. The Python iterators object is initialized using the iter() method. It uses the next() method for iteration.

Answer 3 part ii

```
In [17]: list = [2,4,6,8,10,12,14,16,18,20]
```

In [27]:
```
ch_iterator = iter(list)

print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
```

```
2
4
6
8
10
```

In [28]:
```
string = "abhishek"
```

In [32]:
```
ch_iterator = iter(string)

print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
```

```
a
b
h
i
s
h
e
k
```

answer4 par i

A generator function in Python is a special type of function that allows you to create an iterator, which can be used to iterate over a potentially large sequence of values without storing them all in memory at once. Generator functions use the yield keyword to yield values one at a time during iteration, and they maintain their state between calls.

answer4 par ii

In [44]:
```
def count_up_to(limit):
    n = 1
    while n <= limit:
        yield n
        n += 1

# Create a generator object
counter = count_up_to(5)

# Iterate over the values generated by the generator
for num in counter:
    print(num)
```

```
1
2
3
4
5
```

In [45]:
```python
def total(limit):
    n = 0
    while n <= limit:
        yield n
        n = n+1
counter = total(5)
for i in counter:
    print(i)
```

```
0
1
2
3
4
5
```

In [87]:
```python
# Python3 program to print
# all primes less than N

# Function to check whether
# a number is prime or not .
def isPrime(n):

        # Corner case
        if n <= 1 :
                return False

        # check from 2 to n-1
        for i in range(2, n):
                if n % i == 0:
                        return False

        return True

# Function to print primes
def printPrime(n):
        for i in range(2, n + 1):
                if isPrime(i):
                        print(i, end = " ")
```

In [88]:
```python
printPrime(1000)
```

```
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
107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211
223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331
337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449
457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587
593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709
719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853
857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991
997
```

In [104…

```
def inprime(n):
    if n <=1:
        return False
    for i in range(2,n):
        if n%i == 0:
            return False
    return True

def PrintPrime(n):
    for i in range(2,n+1):
        if inprime(i):
            print(i, end = " ")
```

In [105…

```
PrintPrime(2000)
```

```
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
107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211
223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331
337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449
457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587
593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709
719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853
857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991
997 1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091 1093
1097 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187 1193 1201 1213 1217
1223 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291 1297 1301 1303 1307 1319
1321 1327 1361 1367 1373 1381 1399 1409 1423 1427 1429 1433 1439 1447 1451 1453
1459 1471 1481 1483 1487 1489 1493 1499 1511 1523 1531 1543 1549 1553 1559 1567
1571 1579 1583 1597 1601 1607 1609 1613 1619 1621 1627 1637 1657 1663 1667 1669
1693 1697 1699 1709 1721 1723 1733 1741 1747 1753 1759 1777 1783 1787 1789 1801
1811 1823 1831 1847 1861 1867 1871 1873 1877 1879 1889 1901 1907 1913 1931 1933
1949 1951 1973 1979 1987 1993 1997 1999
```

In [ ]: