Assignment3:

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

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

result = get\_odd\_numbers()

print(result)

2.

\*args allows a function to accept any number of positional arguments as a tuple.

\*\*kwargs allows a function to accept any number of keyword arguments as a dictionary.

Here are examples of functions that demonstrate the use of \*args and \*\*kwargs:

def function\_with\_args(\*args):

for arg in args:

print(arg)

function\_with\_args(1, 2, 3, 4, 5)

def function\_with\_kwargs(\*\*kwargs):

for key, value in kwargs.items():

print(key, value)

function\_with\_kwargs(name="John", age=30, city="New York")

3. In Python, an iterator is an object that allows iteration over a container or sequence of elements. The iterator object must implement two methods:

\_\_iter\_\_() method is used to initialize the iterator object and return itself.

\_\_next\_\_() method is used to retrieve the next element from the iterator.

An example of how to use these methods to print the first five elements of the given list [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]:

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

iterator = iter(my\_list)

for \_ in range(5):

print(next(iterator))

The output will be:

2

4

6

8

10

4. A generator function in Python is a special type of function that uses the yield keyword instead of return to return a sequence of values. It allows you to generate a series of values on-the-fly, without storing them in memory all at once.

Generator functions are defined using the def keyword, like regular functions.

The yield keyword is used in a generator function to produce a value and temporarily suspend the function's execution. When the generator is iterated, the function resumes from where it left off and continues executing until it encounters the next yield statement.

An example of a generator function that generates a sequence of even numbers:

def even\_numbers\_generator():

num = 0

while True:

yield num

num += 2

# Create an instance of the generator

generator = even\_numbers\_generator()

# Use the next() method to retrieve the next value from the generator

print(next(generator)) # Output: 0

print(next(generator)) # Output: 2

print(next(generator)) # Output: 4

# ...

5. Here's a generator function that generates prime numbers less than 1000 and prints the first 20 prime numbers using the next() method:

def prime\_generator():

primes = []

num = 2

while True:

is\_prime = True

for prime in primes:

if num % prime == 0:

is\_prime = False

break

if is\_prime:

primes.append(num)

yield num

num += 1

# Create an instance of the generator

generator = prime\_generator()

# Print the first 20 prime numbers

for \_ in range(20):

print(next(generator))

A6. Here's a Python program that prints the first 10 Fibonacci numbers using a while loop:

a, b = 0, 1

count = 0

while count < 10:

print(a)

a, b = b, a + b

count += 1

7. Here's a list comprehension to iterate through the given string "pwskills" and return a list of certain characters:

string = 'pwskills'

result = [char for char in string if char in ['p', 'w', 's', 'k', 'i', 'l']]

print(result)

The output will be:

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

8. Here's a Python program that checks whether a given number is a palindrome or not using a while loop:

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

temp = number

reverse = 0

while temp > 0:

remainder = temp % 10

reverse = (reverse \* 10) + remainder

temp = temp // 10

if number == reverse:

print(number, "is a palindrome")

else:

print(number, "is not a palindrome")

9. Here's a code using list comprehension to print odd numbers from 1 to 100:

odd\_numbers = [num for num in range(1, 101) if num % 2 != 0]

print(odd\_numbers)

The output will be a list of odd numbers from 1 to 100.