Assignment on Comprehension

1. **Find all of the numbers from 1–1000 that are divisible by 8.**

```
numbers = [x for x in range(1, 1001) if x % 8 == 0]
print(numbers)
```

2. **Find all of the numbers from 1–1000 that have a 6 in them.**

```
numbers_with_6 = [x for x in range(1, 1001) if '6' in str(x)]
print(numbers_with_6)
```

3. **Count the number of spaces in a string (take input from user).**

```
s = input("Enter a string: ")
space_count = s.count(' ')
print("Number of spaces:", space_count)
```

4. **Remove all of the vowels in a string (take input from user).**

```
s = input("Enter a string: ")
vowels = "aeiouAEIOU"
no_vowels = ''.join([char for char in s if char not in vowels])
print(no_vowels)
```

5. **Find all of the words in a string that are less than 5 letters (take input from user).**

```
s = input("Enter a string: ")
short_words = [word for word in s.split() if len(word) < 5]
print(short_words)
```

6. **Use a dictionary comprehension to count the length of each word in a sentence (take input from user).**

```
sentence = input("Enter a sentence: ")
word_lengths = {word: len(word) for word in sentence.split()}
print(word_lengths)
```

7. **Use a nested list comprehension to find all of the numbers from 1–1000 that are divisible by any single digit.**

```
numbers = [x for x in range(1, 1001) if any(x % d == 0 for d in range(1, 10))]
print(numbers)
```

Assignment on Generator

1. We want to generate Fibonacci numbers up to a certain limit. Instead of computing and storing the entire sequence in memory, create generator to yield Fibonacci numbers one by one, conserving memory and allowing for easy iteration.

```
def fibonacci(limit):
    a, b = 0, 1
    while a <= limit:
        yield a
        a, b = b, a + b
for num in fibonacci(100):
    print(num, end=" ")
```

2. Implement a generator function that yields palindrome numbers. Palindromes are numbers that read the same backward as forward (e.g., 121, 1331). Generate palindromes lazily and infinitely.

```
def is_palindrome(n):
    s = str(n)
    return s == s[::-1]

def palindrome_generator():
    n = 0
    while True:
        if is_palindrome(n):
            yield n
        n += 1

gen = palindrome_generator()
for _ in range(10):
    print(next(gen))
```

3. Write a generator function that mimics the behavior of the built-in range() function. The generator should take start, stop, and step arguments and yield numbers within the specified range.

```
def custom_range(start, stop=None, step=1):
if stop is None:
    stop = start
    start = 0
if step == 0:
    raise ValueError("step argument must not be zero")

if step > 0:
    while start < stop:
        yield start
        start += step
else:
    while start > stop:
        yield start
        start += step

for i in custom_range(2, 10, 2):
print(i, end=" ")
```

Assignment of Decorator

1. **Develop a memoization decorator that caches the results of function calls and returns the cached result when the same inputs occur again. This can greatly improve the performance of recursive or computationally intensive functions.**

```
from functools import wraps
def memoize(func):
    cache = {}

    @wraps(func)
    def wrapper(*args, **kwargs):
        key = (args, tuple(sorted(kwargs.items())))
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]

    return wrapper
@memoize
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

print(fib(35))
```