Assignment-25

1) What is the difference between enclosing a list comprehension in square brackets and parentheses?

Ans.

```
# List comprehension (square brackets) - creates a list in memory
list_comp = [x**2 for x in range(1000000)]
print(f"List comprehension type: {type(list_comp)}")
print(f"List comprehension memory: {list_comp.__sizeof__()} bytes")

# Generator comprehension (parentheses) - creates a generator object
gen_comp = (x**2 for x in range(1000000))
print(f"Generator comprehension type: {type(gen_comp)}")
print(f"Generator comprehension memory: {gen_comp.__sizeof__()} bytes")

# Both can be iterated over
for i, value in enumerate(gen_comp):
    if i < 5:
        print(value, end=' ')
    else:
        break
print("\n")
```

Key differences:

i. Memory usage - list comprehension creates all values at once

ii. Reusability - list can be iterated multiple times, generator only once

iii. Access - list allows random access, generator is sequential only

2) **What is the relationship between generators and iterators?**

Ans. Relationship between generators and iterators: Generators are a special type of iterator. All

generators are iterators, but not all iterators are generators.

```
# Custom Iterator
class CountUpTo:
    def __init__(self, max_value):
        self.max_value = max_value
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.max_value:
            self.current += 1
            return self.current
        raise StopIteration

# Generator function
def count_up_to(max_value):
    current = 0
    while current < max_value:
        current += 1
        yield current

# Both achieve the same result
iterator = CountUpTo(3)
generator = count_up_to(3)

print("Iterator output:", end=" ")
for num in iterator:
```

```
    print(num, end=" ")

print("\nGenerator output:", end=" ")
for num in generator:
    print(num, end=" ")
```

Key similarities:

i. Both implement the iterator protocol (__iter__ and __next__)

ii. Both generate values on-demand

iii. Both can be used in for loops

Key differences:

i. Generators use 'yield' keyword for simpler syntax

ii. Generators automatically handle StopIteration

iii. Generators maintain their state automatically

3) What are the signs that a function is a generator function?

Ans. Signs that a function is a generator function:

- It contains at least one yield statement
- Calling the function returns a generator object, not the actual values
- Using the function in a for loop runs until there are no more values to yield

4) What is the purpose of a yield statement?

Ans. Purpose of a yield statement:

```
def number_generator(n):
    print("Starting...")
    for i in range(n):
        print(f"About to yield {i}")
        yield i
```

```
        print(f"After yielding {i}")

# Using the generator
gen = number_generator(3)
print(f"Generator object created: {gen}")

print("\nIterating:")
for num in gen:
    print(f"Received {num}")
```

Key points about yield:

i. Pauses function execution and returns a value

ii. Maintains function state between calls

iii. Allows for memory-efficient iteration

iv. Creates a generator object when the function is called

5) What is the relationship between map calls and list comprehensions? Make a comparison and contrast between the two.

Ans. Relationship between map calls and list comprehensions:

Key comparisons:

i. Readability: list comprehensions often more readable

ii. Performance: generally similar

iii. Memory: map returns iterator, list comp creates list

iv. Functionality: list comp can do filtering in one line

When to use map:

- Working with existing functions

- Functional programming style

- When you need an iterator

When to use list comprehension:

- More complex operations
- When you need a list immediately
- When combining mapping and filtering