Generators

Python generators are a simple way of creating iterators.

- Generator doest have `return` statement instead they have `yield`

```
def gen_demo():

    yield 'first'
    yield 'second'
    yield 'third'
```

```
gen = gen_demo()

for i in gen:
        print(i)

# first
# second
# third
```

yield vs return

Feature	Return	Yield
Function Termination	Terminates	Pauses
Value Return	Single Value	Multiple Value
Object Returned	Value	Generator object
Execution Flow	Linear	Pauses and resumes

```
def numbers(n):
    for i in range(n):
        yield i

for num in numbers(5):
    print(num)  # Output: 0 1 2 3 4
```

```
def square(x):
    return x * x

result = square(5)
print(result)  # Output: 25
```

When to Use Which

- **Return:** Use `return` when you want to compute a single value and return it to the caller.
- **Yield:** Use `yield` when you want to generate a sequence of values over time, without storing them all in memory at once. This is particularly useful for large datasets or infinite sequences.

In essence:

- **Return** is like giving a complete gift at once.
- **Yield** is like giving a gift in small, wrapped packages, one at a time, with pauses in between.

Range Function using generator

```
def mera_range(start, end):

        for i in range(start, end):
                yield i
```

```
gen = mera_range(1, 5)

for i in gen:
            print(i)

# 1
# 2
# 3
# 4
```

Generator Expression

```
L = [i**2 for i in range(1,4)]

gen = (i**2 for i in range(1,4))

for i in gen:
        print(i)

# 1
# 2
# 3
```

Benefits of using a Generator

1. Memory Efficient
2. Ease of Implementation
3. Representing Infinite Streams
4. Chaining Generators