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

* **Square Brackets []**:
  + Creates a **list**.
  + Example:

squares = [x\*\*2 for x in range(5)]

print(squares) # Output: [0, 1, 4, 9, 16]

* **Parentheses ()**:
  + Creates a **generator expression**.
  + Example:

squares = (x\*\*2 for x in range(5))

print(squares) # Output: <generator object <genexpr> at 0x...>

**Difference**:

* List comprehensions ([]) generate the entire list in memory immediately.
* Generator expressions (()) produce items on-the-fly, saving memory.

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

* **Generators**:
  + A type of iterator created using functions with yield or generator expressions.
  + Example:

def my\_generator():

yield 1

yield 2

* **Iterators**:
  + Objects that implement the iterator protocol (\_\_iter\_\_() and \_\_next\_\_()).
  + Example:

class MyIterator:

def \_\_init\_\_(self):

self.current = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.current < 2:

self.current += 1

return self.current

raise StopIteration

**Relationship**:

* Generators are a convenient way to create iterators.
* Both generators and iterators allow iteration over a sequence of values.

**3. What are the signs that a function is a generator function?**

* **Use of yield**:
  + A function containing the yield keyword is a generator function.
  + Example:

def my\_generator():

yield 1

yield 2

* **Returns a generator object**:
  + Calling a generator function returns a generator object, not a value.
  + Example:

gen = my\_generator()

print(gen) # Output: <generator object my\_generator at 0x...>

**4. What is the purpose of a yield statement?**

* **Purpose**:
  + Pauses the function's execution and returns a value to the caller.
  + When the generator is resumed, execution continues from where it was paused.
  + Example:

def my\_generator():

yield 1

yield 2

* **Use Case**:
  + Used in generator functions to produce a sequence of values lazily (on-the-fly).

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

* **map**:
  + Applies a function to all items in an iterable.
  + Returns an iterator of results.
  + Example:

result = map(lambda x: x\*\*2, [1, 2, 3])

print(list(result)) # Output: [1, 4, 9]

* **List Comprehension**:
  + Creates a list by applying an expression to each item in an iterable.
  + Example:

result = [x\*\*2 for x in [1, 2, 3]]

print(result) # Output: [1, 4, 9]

**Comparison**:

* Both map and list comprehensions transform items in an iterable.
* Both can achieve similar results.

**Contrast**:

* **Readability**:
  + List comprehensions are often more readable and concise.
* **Flexibility**:
  + List comprehensions can include conditions (e.g., [x\*\*2 for x in [1, 2, 3] if x > 1]).
  + map requires filter for conditional logic.
* **Performance**:
  + List comprehensions are generally faster for small datasets.
  + map can be more memory-efficient for large datasets since it returns an iterator.