Task 1: Working with List Comprehensions

Objective:

To understand and practice list comprehension by solving problems using this Python feature.

Steps:

1. **Create a list of squares**: Write a list comprehension to generate a list of squares for numbers 1 through 10.
2. **Filter even numbers**: Modify the list comprehension to only include the squares of even numbers.
3. **Create a list of tuples**: Write a list comprehension to create a list of tuples, where each tuple contains the number and its square for numbers from 1 to 10.
4. **Flatten a nested list**: Given a 2D list, use list comprehension to flatten it into a 1D list.

Example:

```
# List comprehension to generate squares of numbers 1 to 10
squares = [x ** 2 for x in range(1, 11)]
print(squares)
```

Questions:

1. What are the benefits of using list comprehension over traditional `for` loops?
2. How can you add conditions to a list comprehension? Provide an example.
3. How would you modify a list comprehension to create a list of cubes instead of squares?

Task 2: Understanding Lambda Functions

Objective:

To understand how lambda (anonymous) functions work and practice applying them in Python.

Steps:

1. **Basic Lambda Function**: Write a lambda function that takes two arguments and returns their sum.
2. **Lambda with `map()`**: Use a lambda function with `map()` to multiply each element in a list by 2.
3. **Lambda with `filter()`**: Use a lambda function with `filter()` to get all even numbers from a given list of numbers.
4. **Lambda with `sorted()`**: Write a lambda function to sort a list of tuples by the second element.

Example:

```
# Lambda function to add two numbers
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8
```

Questions:

1. How does a lambda function differ from a regular function defined using `def`?
2. Can you use a lambda function without `map()`, `filter()`, or `sorted()`? If so, give an example.
3. What would happen if you used a lambda function with multiple expressions? Can you do it?

Task 3: Exploring Iterators

Objective:

To practice creating and using iterators in Python.

Steps:

1. **Create a custom iterator class**: Define a class `Range` that behaves like the built-in `range()` function. The class should generate numbers from `start` to `end` using an iterator.
2. **Iterate through a list using an iterator**: Write a program to create an iterator from a list of numbers and use it to print all elements of the list.
3. **Handle StopIteration**: Implement a `while` loop to manually iterate over an iterator and handle the `StopIteration` exception.

Example:

```
# Simple iterator using a list
numbers = [10, 20, 30]
iterator = iter(numbers)
print(next(iterator))  # Output: 10
```

Questions:

1. What is the difference between an iterator and a list?
2. How does the `StopIteration` exception work in iterators? Give an example of its usage.
3. Why is the `__iter__()` method needed in an iterator class?

Task 4: Working with Generators

Objective:

To understand and implement generators in Python.

Steps:

1. **Create a simple generator**: Write a generator function `count_up_to(n)` that yields numbers from 1 to `n`.
2. **Generator for Fibonacci**: Create a generator function that yields the first `n` numbers of the Fibonacci sequence.
3. **Generator Expression**: Create a generator expression that generates squares of all numbers from 1 to 10 and print them one by one.
4. **Use of `next()` with generators**: Use the `next()` function to manually retrieve values from a generator function.

Example:

```
# Simple generator function
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

# Using the generator
gen = count_up_to(5)
for num in gen:
    print(num)
```

Questions:

1. How does the `yield` statement differ from `return` in a function?
2. Can you explain the memory efficiency advantages of using generators over regular functions or lists?
3. What is the role of `next()` when working with generators? How do you use it?

Task 5: Combine List Comprehensions, Lambdas, Iterators, and Generators

Objective:

To practice combining list comprehensions, lambda functions, iterators, and generators in a single program.

Steps:

1. **Use a generator to generate numbers**: Create a generator that yields numbers from 1 to 10.
2. **Apply a lambda to the generator**: Use a lambda function with `map()` to square each number from the generator.
3. **Convert the generator to a list**: Use list comprehension to convert the results of the `map()` function to a list.
4. **Filter the results**: Use a lambda function with `filter()` to get only the even numbers from the list.

Example:

```
# Generator function to generate numbers
def generate_numbers(n):
    for i in range(1, n+1):
        yield i

# Using lambda with map and filter
gen = generate_numbers(10)
squares = list(map(lambda x: x ** 2, gen))
even_squares = list(filter(lambda x: x % 2 == 0, squares))

print(even_squares)  # Output: [4, 16, 36, 64, 100]
```

Questions:

1. How can you combine `map()` and `filter()` with list comprehensions?
2. Why would you use generators in combination with lambda functions instead of regular lists?
3. Can you think of a real-world scenario where combining these techniques would be useful?