Advanced Python Workbook: Iterables, Iterators, and Generators

This workbook is designed to bridge the gap between intermediate and advanced Python programming. By the end of these exercises, you will understand how Python handles data streams, manages memory via lazy evaluation, and implements the iterator protocol.

1. Iterables – Basics

Exercise 1 (Simple)

Goal: Distinguish between an iterable and an iterator.

```
# Use the dir() function or isinstance from collections.abc to check
# if the following objects are Iterables or Iterators.
# Objects: [1, 2, 3], "Python", (x for x in range(5))

items = [1, 2, 3]
name = "Python"
gen = (x for x in range(5))

# Your code here:
# print(f"List is iterable: {???}")
# print(f"Gen is iterator: {???}")
```

Exercise 2 (Simple)

Goal: Manual iteration. Convert the list `['a', 'b', 'c']` into an iterator and retrieve all elements using `next()`. Handle the `StopIteration` exception manually.

Exercise 3 (Medium)

Goal: Multiple Iterators. Create a list `data = [10, 20]`. Create two separate iterator objects from this same list. Call `next()` on the first one once, then `next()` on the second one twice. Explain the output.

Exercise 4 (Hard)

Goal: The `iter()` sentinel pattern. The `iter(callable, sentinel)` form is rarely used but powerful. Create a function `produce_data()` that returns a random integer between 1 and 10. Use `iter()` to create an iterator that stops when the function produces the number `7`.

2. Iterators – Built-in & Custom

Exercise 5 (Simple)

Goal: Custom Iterator Protocol. Complete the `SquareIterator` class to return squares of numbers from 1 to `n`.

```
class SquareIterator:
    def __init__(self, n):
        self.n = n
        self.current = 1

    def __iter__(self):
        return ???

    def __next__(self):
        if self.current <= self.n:
            res = self.current ** 2
            self.current += 1
            return res
        else:
            ???
```

Exercise 6 (Medium)

Goal: Infinite Iterators. Use `itertools.count` to create an iterator starting at 10, incrementing by 5. Print the first 5 values using a loop and a break condition.

Exercise 7 (Medium)

Goal: Iterator Resetting. Does a custom iterator reset if you loop over it a second time? Create an instance of your `SquareIterator` from Exercise 5, loop over it once, then try to loop over the same instance again. Observe the behavior.

Exercise 8 (Hard)

Goal: The "Cycler". Create a class `CircularIterator` that takes a list and repeats its elements indefinitely (e.g., `[1, 2] -> 1, 2, 1, 2...`). Do not use `itertools.cycle`.

3. Generators – Functions and Expressions

Exercise 9 (Simple)

Goal: First Generator. Write a generator function `countdown(n)` that yields numbers from `n` down to 1.

Exercise 10 (Simple)

Goal: Generator Expression. Convert this list comprehension into a generator expression: `[x * 2 for x in range(1000)]`.

Exercise 11 (Medium)

Goal: Yielding from a loop. Create a generator `step_range(start, stop, step)` that mimics the behavior of `range()` but allows float increments.

Exercise 12 (Hard)

Goal: Generator State. Write a generator `running_average()` that allows you to `.send()` numbers to it and yields the current average. *Hint: You will need to use* `(yield total/count)`.

4. Memory & State Management

Exercise 13 (Simple)

Goal: Size Comparison. Use `sys.getsizeof()` to compare a list of 1 million integers vs. a generator producing 1 million integers.

Exercise 14 (Medium)

Goal: Lazy Filtering. You have a "dirty" list of strings: `[" apple", "orange ", " ", "banana", "STOP", "cherry"]`. Create a generator pipeline that:

1. Strips whitespace.
2. Filters out empty strings.
3. Stops entirely if it encounters the string "STOP".

Exercise 15 (Hard)

Goal: Large File Processing (Simulation). Simulate a 10GB log file by creating a generator that yields 1 million lines of text. Write a second generator that only yields lines containing the word "ERROR". Measure memory usage during iteration.

5. Combined Concepts & Advanced Use Cases

Exercise 16 (Medium)

Goal: `yield from`. Use `yield from` to flatten a nested list: `[[1, 2], [3, 4], [5]]`.

Exercise 17 (Hard)

Goal: Chaining Generators. Create three generators:

1. `integers()`: yields 1, 2, 3...
2. `squared(seq)`: takes a sequence and yields squares.
3. `negated(seq)`: takes a sequence and yields negative values. Chain them to produce `-1, -4, -9, -16...`

6. Paragraphic / Scenario-based Hard Exercises

Exercise 18: The Infinite Fibonacci Stream Design a solution that generates Fibonacci numbers indefinitely. However, the consumer should be able to request the "next" number at any time without the program recalculating the entire sequence. The memory footprint must remain constant regardless of how many numbers have been generated.

Exercise 19: The Log File Tailer Imagine a log file that is being actively written to by another process. Create a Python generator that "tails" this file (like the Unix `tail -f` command). It

should wait for new lines to be added and yield them as they appear. If no new line is found, it should sleep briefly before checking again.

Exercise 20: CSV Batch Processor You have a CSV file with 1 million rows. You need to process these rows in batches of 500 to upload to a database. Create a generator that takes an iterable (the file rows) and yields lists (batches) of a specified size.

Exercise 21: Data Pipeline Validation Create a pipeline of generators for a sensor data stream. The first generator simulates sensor readings (floats). The second generator clips values to a range [0, 100]. The third generator calculates a rolling window sum of the last 3 values.

Exercise 22: The "Smart" Page Crawler Create a generator that simulates crawling a website. It takes a list of URLs. It yields the HTML length of each page. If a URL fails, it should catch the error and yield `None`, but continue to the next URL. Use `.send()` to allow the user to inject a new URL into the crawl queue mid-iteration.

Exercise 23: File System Searcher Write a generator that recursively walks through a directory tree and yields the full path of every `.py` file it finds. It must be memory efficient (don't build the whole list first).

Exercise 24: Interleaved Streams You have two sorted lists of timestamps. Write a generator that merges these two iterables into a single sorted stream without ever loading both lists fully into memory.

Exercise 25: Windowed Iterator Create a function `window(iterable, size)` that returns a sliding window of the input. For `[1, 2, 3, 4]` and `size=2`, it yields `(1, 2), (2, 3), (3, 4)`.

Exercise 26: Unique Filter Generator Write a generator that takes an iterable and yields only the unique elements in the order they appear. It must handle infinite input streams (so it keeps track of what it has seen).

Exercise 27: The Round-Robin Scheduler You have three lists of tasks: `Low`, `Medium`, and `High` priority. Create a generator that yields one task from each list in a round-robin fashion until all lists are exhausted.

7. Solutions

Section 1 & 2: Iterables & Iterators

Sol 1:

```
from collections.abc import Iterable, Iterator
items = [1, 2, 3]
gen = (x for x in range(5))
print(isinstance(items, Iterable)) # True
print(isinstance(items, Iterator)) # False (Lists are not their own iterators)
print(isinstance(gen, Iterator))   # True
```

Tricky Part: An **Iterable** is something you can get an iterator from (has `__iter__`). An **Iterator** is the object that actually tracks state and has `__next__`.

Sol 5:

```
class SquareIterator:
    def __init__(self, n):
        self.n = n
        self.current = 1
    def __iter__(self):
        return self
    def __next__(self):
        if self.current <= self.n:
            res = self.current ** 2
            self.current += 1
            return res
        raise StopIteration
```

Section 3 & 4: Generators

Sol 12 (Running Average):

```
def running_average():
    total = 0
    count = 0
    avg = None
    while True:
        num = yield avg
        if num is None: break
        total += num
        count += 1
        avg = total / count

g = running_average()
next(g) # Prime the generator
print(g.send(10)) # 10.0
print(g.send(20)) # 15.0
```

Tricky Part: You must "prime" a generator with `next()` or `g.send(None)` before you can send data to a `yield` expression.

Section 6: Hard Scenarios

Sol 18 (Fibonacci):

```
def fibonacci_gen():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci_gen()
```

```
# Use: next(fib) -> 0, next(fib) -> 1...
```

Explanation: Generators are perfect for infinite sequences because they only calculate the "next" value on demand, using minimal memory (just storing `a` and `b`).

Sol 20 (Batching):

```
def batcher(iterable, size):
    it = iter(iterable)
    while True:
        batch = []
        try:
            for _ in range(size):
                batch.append(next(it))
            yield batch
        except StopIteration:
            if batch: yield batch
            break
```

Common Mistake: Forgetting to yield the last partial batch if the total count isn't perfectly divisible by the batch size.

Sol 25 (Sliding Window):

```
from collections import deque
def window(iterable, size):
    it = iter(iterable)
    d = deque(maxlen=size)
    # Fill initial window
    for _ in range(size):
        d.append(next(it))
    yield tuple(d)
    for item in it:
        d.append(item)
        yield tuple(d)
```

Key Takeaways for the Learner

1. **Iterables** define `__iter__`. **Iterators** define `__iter__` AND `__next__`.
2. **Generators** are a subset of Iterators. They save their local state automatically.
3. **Lazy Evaluation:** Elements are produced only when requested. This allows for processing datasets larger than RAM.
4. **Exhaustion:** Iterators and Generators are "one-way." Once you iterate to the end, they are empty. You must create a new instance to start over.