Lauryn Curtiss ☺

Logan Rook ☺

Amari Robinson

Wade Ratzlaff

Practice Lab: Iterators, Generators, yield, and Dunder Methods

Lauryn's Observations:

Part 1: In this cell it defines a list called sensor_readings that contains ten values in it and it uses a standard for loop to print them all. The entire dataset is processed and displayed in one go.

Part 2: In this second cell it moves towards manual iteration by its use of the iter() function to make a iterator object from the sensor_readings list by then calling next(it) three times it demonstrates that the iterator keeps it state and retrieves the value.

Part 3: In the third cell it implements a generator function using the yield keyword which then allows the code to pause in what its doing and then resume its execution. The output will only be given when next stream is particularly called showing that the values are generated on demand rather than being pre called.

Amari's Observations:

Part 4:

When I run the normal function I noticed that it prints "Start" and then it stops as soon as it reaches the return statement, so anything after that never ran. In the generator function, I see that the function pauses each time it reaches a "yield". Each call to next() continues the function from where it all left off, allowing it to print messages and return values step by step instead of all at once.

Part 5:

When I run this code, I see that the generator starts at 0 and increases by 1 each tim eI call next(). The counter function does not end because of the while true loop, so it keeps yielding the next number instead of stopping.

Logan's Observations:

Part 6: This code snippet here pulls data from the sensor_stream, and only yields a result which meets the high_temp_filter's criteria (being over the threshold, which is 23.0 in our pipeline). The first yield is 23.1, but we have next(filtered) run twice, and the pipeline continues where it

left off, yielding 23.4. It is interesting to observe that the second next(filtered) continues after the 23.1, and not starting fresh at the beginning of the dataset.

Part 8: We can observe that iter(gen) is gen is true, and that we dont need to __iter__ because generators are already iterators.

Wade's Observations:

Part 7.1: First we define the class User. The User class started with __init__ defining self names like name and age. It passes it to __str__ which tells it it needs to go into a string. Then it defines __repr__ to represent the official string as an object. We attach name u to the class User and print name and age. User, "Alice", 21, name, and age are part of heap. User("Alice", 21), u, __init__ is part of stack.

Part 7.2: The code defines a class called Cart. The class creates the functions using dunder methods __init__(defines the self and item object), __len__(counts the number of items in cart), and __bool__(which makes sure the cart actually has items in it by putting >0). Then we ask if the cart exists, if it does print the items in cart. Cart, the list, and the strings "apple" and "banana" are part of heap. __init__ and the name cart are part of stack.

Part 7.3: Here we have an example of duck typing. __getitem__ stores the data from the list so that when log(0) or log(2) runs it knows to pull the first and third item from the list(10 and 30 respectively). So when we run the for loop, it can go straight to the log name and iterate over the list one by one(10 through 40). Log, list, and the integers are part of heap. __init__ call and the name log are part of stack.

Part 7.4: Here checks the first value and sees if it equals the other value using the __eq__ dunder method. Then it creates another function that checks if the first value is less than the second value with __lt__. Then it uses __add__ which tells the class it needs to add the two values. All of these help it treat Score as integers we can compute with. Score object and the integers 10 and 20 are part of heap. Names a and b and the __init__ call are stack.

LAB 3 FRIDAY WORK:

Part 1:

• Where is x stored? - x is stored in the heap because it is an object.

• Why does x still exist? - Because x never needs to get destroyed. x is kept alive in a closure because the inner function keeps x alive while outer finishes. Outer is assigned to x so that f() can print x.

• Does execution restart each call? - Yes and no. The f() starts the call again so this restarts the function inner. Inner is on the stack so it gets destroyed and restart. What doesn't get rewritten is the object x=10. This is why the output for f() prints the same x.

• Why doesn't count reset to 0? - This is because the increment function tells count to increase by one each time it is called again. It increases by one each time because count doesn't get destroyed.

• Where does count live after counter() returns? - Count lives on the heap because counter returns.

Part 2.1

• Which function runs first?

My_decorator runs first (at definition time)

• Did we modify greet()?

The original greet() isnt modified, but the name greet points to the wrapper now.

• What code added the extra behavior?

The wrapper did.

Part 2.2-

• Decorator runs once at definition time

Yes, the decorator wraps itself immediately.

• Wrapper runs every time the function is called

Greet is now a reference to the wrapper function.

• Original function runs only if the wrapper calls it

Yes, the original function is trapped in the wrapper functions.

Part 2.3-

• The function name now refers to the wrapper

Yes, python reasigns the name "say_hi" to point to the wrapper.

• The original function is stored in the wrapper's closure

Yes, when you call the original function youre actually opening the wrapper.

Part 3-

Where is logging added?: The logging is in this line "print("Calling with args:", args)" as this line runs before the function even executes so you see the arguments logged first.

Did add change?: No the add function itself did not change

Why do we need *args?: *Args is flexible as it accepts any number of positional arguments so in this context of the code we need it for the part of the function where it return the value for a+b in this context one of them being 3+4 which with *args returns 7.

Part 4.1-
In this example, the closure is used to store data. When make_multiplier(n) is called, the value of n is remembered by the inner function multiply, even after make_multiplier has finished running. Each time multiply is called, it still has access to its own saved value of n. This is why double(5) and triple(5) produce different results. There is no wrapper involved and no extra behavior added. The closure's only role here is preserving data across function calls.

Part 4.2-
In this example, the decorator adds behavior to an existing function. The loud function wraps the original speak function inside a wrapper function. When speak() is called, the wrapper runs first and prints "LOUD MODE," then calls the original function. The code inside speak is not changed, but its behavior is extended. This shows how decorators modify how a function runs by wrapping it rather than by storing data.