5.8 PythonTA and Accumulation Tables

Accumulation tables are a powerful tool for understanding the behaviour of for loops, but they also can be a bit time-consuming to create when given large quantities of data, or for more complex loops. So to make it easier to use accumulation tables as part of your code-writing process, PythonTA provides a way of generating and displaying loop accumulation tables!

Let's illustrate using PythonTA in this way by modifying our `product` function from 5.4 Repeated Execution: For Loops.

```
from python_ta.debug import AccumulationTable  # New impor


def product(numbers: list[int]) -> int:
    """Return the product of the given numbers.

    >>> product([10, 20])
    200
    >>> product([-5, 4])
    -20
    """
    # ACCUMULATOR product_so_far: keep track of the produc
    # of the elements in numbers seen so far in the loop.
    product_so_far = 1

    with AccumulationTable(['product_so_far']):  # New lin
        for number in numbers:
            product_so_far = product_so_far * number

    return product_so_far
```

Now, here's what happens when we call `product` in the Python console:

```
>>> product([10, 20, 30, 40])
iteration    number    product_so_far
-----------  --------  ---------------
0            N/A       1
1            10        10
2            20        200
3            30        6000
4            40        240000
240000
```

The very last line, containing just `240000`, is the function's return value, as we would normally expect to see when calling a function in the Python console. The part above it is what's interesting: a text-based representation of an accumulation table, displaying the loop iteration number (0 to 4), the value of the loop variable `number` at each iteration, and the value of the accumulator `product_so_far`.

Pretty cool! Now let's take a few moments to discuss how this new code is structured.

Breaking down the code

This code imports `AccumulationTable`, a *custom data type* defined by PythonTA. What does this data type do, exactly? When we write `AccumulationTable(['product_so_far'])`, we are creating a new Python value that will expect to "wrap around" a for loop and keep track of the accumulator variable `product_so_far` at each loop iteration.

We achieve this "wrap around" behaviour through a new form of Python syntax called a **with statement**:[1]

[1] Yet another form of compound statement, like if statements and for loops!

```
with <value>:
    <statement1>
    <statement2>
    ...
```

For our purposes, with statements are similar to decorators:

- A with statement "wraps around" a block of code, just like a decorator "wraps around" a function or class definition.
- A with statement "modifies the behaviour" of a block of code, just like a decorator "modifies the behaviour" of a function or class definition.
- You aren't responsible for knowing the technical details of how with statements or decorators actually work, just what they do.

In general, with statements offer programmers the ability to modify the behaviour of a block of code. Using `with AccumulationTable(['product_so_far']):` to wrap the for loop, the Python interpreter does the following:

1. Evaluate `AccumulationTable(['product_so_far'])`, producing a value of the `AccumulationTable` data type that is intended to track the accumulator variable `product_so_far`.
2. Then execute the for loop written inside the with statement.
    - Immediately before the for loop executes, and at the end of each iteration of the for loop, the `AccumulationTable` value records the iteration number, value of the loop variable, and value of the `product_so_far` accumulator.
3. At the end of the with statement, after the for loop ends, the `AccumulationTable` *prints* a text-based table of the values it recorded in Step 2.

A subtlety: printing vs. return value

One subtlety with `AccumulationTable` is that even though it records data in a tabular form, it *prints* the table to the Python console, rather than making that data available in code. To see what we mean by this, let's redo our Python console interaction, but now use an assignment statement to store `product`'s return value:

```
>>> result = product([10, 20, 30, 40])
iteration    number    product_so_far
-----------  --------  ---------------
0            N/A       1
1            10        10
2            20        200
3            30        6000
4            40        240000
```

Now, the accumulation table is still displayed immediately, but the return value `240000` isn't displayed below—it's be stored in the `result` variable, just as we would expect in any other assignment statement.

Let's see what happens when we ask the Python interpreter to evaluate `result`:

```
>>> result
240000
```

The return value (`240000`) is displayed, but the accumulation table isn't part of that return value. *Using `AccumulationTable`, we can see the accumulation table for a given for loop, but we cannot access or compute with the underlying data, just look at the result.*[2] This illustrates one of the key limitations of "printing" data: we see that data immediately in the Python console, but can't access it directly. In CSC110/111, we'll use printing techniques like `AccumulationTable` as a way to gain information about our code, which is especially helpful when debugging. However, printing should never replace the usual data flow of storing the results of computations in values and returning values from functions! And one final warning:

[2] Okay, technically there is a way to access the data recorded by the `AccumulationTable`, but that's beyond the scope of this course. Please feel free to ask about this during office hours!

> **Warning**: `python_ta.debug.AccumulationTable` is a useful tool for debugging purposes, and we hope you make use of it when completing your work for CSC110/111. However, because the accumulation table prints debugging text to the Python console, it violates our the principle of function correctness: *a function's body should do exactly what the function specifcation says, no more and no less.*
>
> That means that before submitting any graded work for this course, you should always *remove* the with statement containing `AccumulationTable` (and unindent the for loop contained within) before your final submission!