LOOP PATTERNS

The Python Bakery

Many Loop Patterns

- Count
- Sum
- Accumulate
- Map
- Filter

There are so many ways to use for loops.
In this lesson, we will see a few common patterns.
These patterns shown here are just starting points.
Think of them as templates that can be adapted and combined to solve more complex problems.

The Count Pattern

```
a_list = ["Alpha", "Beta", "Gamma"]

count = 0
for item in a_list:
    count = count + 1

print(count)
```

We have a list of items, and want to know how many there are.
A simple algorithm is, starting with an initial value of 0, to add 1 for each element we see to a count variable.
When the loop is finished, the "count" variable will have the length of the list.

The Sum Pattern

```
a_list = [10, 30, 20]

sum = 0
for item in a_list:
    sum = sum + item

print(sum)
```

We have a list of numbers, and want to add them all up.
The plus operator can only take two items at a time, however.
Therefore, we add each element one at a time to a "sum" variable, which is also initialized to 0.
As you can see the, the sum pattern is similar to the Count pattern, except instead of adding 1, we are adding the iteration variable.

Accumulator Pattern

```
result = ___
for item in a_list:
    result = result ___ item
```

The Sum and Count patterns are both more specific examples of the accumulator pattern.
In general, this pattern allows us to start with an initial value and use any function or operation that takes in two values.
This process of accumulation is also sometimes known as "reducing" or "folding" a list.
This pattern can be applied to numbers, but it also works for strings, booleans, and any type that can be combined with an operator that takes two operands.
You have to think critically about what the initial value should be, and what operator you will use to combine them.

String Accumulation

```
colors = ["A1", "B2", "C3"]

joined = ""
for color in colors:
    joined = joined + color

print(joined)
```

Technically, string accumulation is no different than the summation pattern.
The only difference is that strings are being added together instead of integers.
The end result is a single string with all the source strings joined together.

Boolean Accumulation (any/all)

```
answers = [True, True, False, True]

# Any True?
any_true = False
for answer in answers:
    any_true = any_true or answer
print(any_true)

# All True?
all_true = True
for answer in answers:
    all_true = all_true and answer
print(all_true)
```

The Boolean Accumulation patterns come in two versions: the "any" and the "all".
With "any", we start with an initial value of False.
Then we check if either the current element is True or if the accumulation variable is True.
If at any point, we encounter a True value in the list, then the accumulation variable becomes True and will evaluate to True in every subsequent check.
On the other hand, the "all" variant starts off with the initial value of True.
Each check is whether the current element is True and if the accumulation variable is still True.
If at any point, we encounter a False value in the list, then the accumulation variable becomes False and will evaluate to False in every subsequent check.

Boolean Accumulation in Action

```
from bakery import assert_equal

def any_red(colors: list[str]) -> bool:
    result = False
    for color in colors:
        result = result or color == "red"
    return result

assert_equal(any_red(["blue", "red", "green"]), True)
assert_equal(any_red(["r", "g", "b"]), False)
```

Boolean accumulation is more powerful than it appears at first.
The key insight is that the pattern works on more than just a list of boolean values.
In particular, you can use an expression that produces a boolean in order to ask a question about each element of a list.
Note that no if statement is required; we are simply using or to combine logical conditions together and return the final boolean result.

The Map Pattern

```
old_list = ["Apple", "Grape", "Orange"]

copied_list = []
for item in old_list:
    copied_list.append(item)

print(copied_list)
```

What happens when we accumulate a list?
If we start with an empty list as our initial value, and append each value one at a time, we end up with a copy of the original list.
We could then make further modifications to the new list, without worrying about changing the original list.

Modifying the Map Pattern

```
fruits = ["Apple", "Grape", "Orange"]

plural_fruits = []
for fruit in fruits:
    plural_fruits.append(fruit + "s")

print(plural_fruits)
```

As we're appending values, we can also modify them.
For example, you could double each value from the old list, or convert each temperature from Fahrenheit to Celsius.
In this program, we are simply adding the string "s" to the end of each fruit to make the plural version of the word.
However, any valid operation or expression can be used as arguments to the append method, allowing us to transform each element of the list.

Accidental Infinite Loop

```
fruits = ["Apple", "Grape", "Orange"]

plural_fruits = []
for fruit in fruits:
    # Uh oh, incorrect accumulation variable!
    fruits.append(fruit + "s")

print(plural_fruits)
```

Do you see the mistake in the program shown here?
If you tried running this, you would find your program taking a very long time.
In fact, it might take forever!
The problem is that we are appending to the very same list that we are iterating over.
If you append to the fruits list while you iterate through the fruits list, the for loop will keep finding new elements to process.
Python is more than happy to keep going around in circles until you either stop the program or the heat death of the universe at the end of time, whichever comes first.

The Filter Pattern

```
numbers = [50, 100, 200]

# Filter+Map
new_list = []
for number in numbers:
    if number < 100:
        new_list.append(number)
print(new_list)
```

We have a list of numbers, and want to ignore some of them according to a rule.
By embedding an IF statement inside the loop, we can optionally include or not include elements in our accumulation.
This is the Filter pattern, which is really more of an optional modification we can make to our other patterns.
Here we see the Filter pattern combined with the Map pattern.
Tracing the flow of this program can be a little tricky, but keep in mind that it's the same rules as we've seen before.
The body of the for loop is repeated however many times there are items in the list.
Each time, the if statement's condition is considered, and if True then its body with the append statement is executed.
When the for loop is entirely done, we execute the print statement after the for loop.

The Filter Pattern with Count

```
numbers = [50, 100, 200]

# Filter + Count
result = 0
for number in numbers:
    if number < 100:
        result = result + 1
print(result)
```

The Filter pattern is very compatible with the other patterns.
Here we see a variant of the pattern being combined with the Count pattern.
The basic code structure is the same, with an if statement inside of our for loop.
The big different comes in the initial value and accumulation step, which match the Count pattern instead of the Map pattern.

Inside or Outside of the Body

```
# Before
for item in a_list:
    # Inside
    pass
# After
```

Novices struggle with what goes inside or outside of a loop body.
Remember, every statement inside the body is executed for each element.
Only put things inside if they should happen for each element.
The patterns can help you keep track of where things go, but ultimately you have to think critically to know.

Patterns in Functions

```
from bakery import assert_equal

def count_3s(numbers: list[int]) -> int:
    total = 0
    for number in numbers:
        if number == 3:
            total = total+1
    return total

assert_equal(count_3s([1, 2, 3]), 1)
assert_equal(count_3s([3, 3, 3, 3]), 4)
```

Often, you will use these patterns inside of the functions that consume lists.
All of the same rules about indentation and scope apply as before, so in theory there are no surprises.
However, novices sometimes struggle to put these complex pieces together.
In particular, they are not clear where the return statement goes relative to the body of the for loop.
In this example program, you can clearly see that the return statement should go AFTER the body of the for loop, OUTSIDE of the if statement, but INSIDE the body of the function definition.