5.6 Index-Based For Loops

We have learned a lot about collections so far:

1. We can access the elements of a collection via indexing (e.g., for lists and strings) or key lookups (e.g., for dictionaries).
2. We can evaluate an expression for each element of a collection with comprehensions to produce a new collection.
3. We can execute a sequence of statements for each element of a collection with a for loop.

The loops we have studied so far are *element-based*, meaning the loop variable refers to a specific element in the collection (or key in the dictionary). Though these loops are powerful, they have one limitation: they process each element of the collection independent of where they appear in the collection. In this section, we'll see how we can loop through elements of index-based collections while keeping track of the current index. Looping by index enables us to perform more computations than looping by element alone, because we'll be able to take into account *where* a particular element is in a collection in the loop body.

As usual, before proceeding please take a moment to review the basic loop accumulator pattern:

```
<x>_so_far = <default_value>

for element in <collection>:
    <x>_so_far = ... <x>_so_far ... element ...  # Somehow combine loop variable and accumulator

return <x>_so_far
```

Remembering the problem: repeating code

When we introduced for loops, we presented a `my_sum` implementation that showed the exact statement that is repeated:

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

    >>> my_sum([10, 20, 30])
    60
    """
    sum_so_far = 0

    sum_so_far = sum_so_far + numbers[0]
    sum_so_far = sum_so_far + numbers[1]
    sum_so_far = sum_so_far + numbers[2]

    return sum_so_far
```

Our eventual solution to the `my_sum` function used a loop variable, `number`, in place of the `numbers[_]` in the body. There is another solution if we observe that the indexes being used start at `0` and increase by one on each iteration of the loop. On the last iteration, the index should be `len(numbers) - 1`. This sequence of numbers can be expressed using the `range` data type: `range(0, len(numbers))`. Based on this, let us use a different kind of for loop to implement `my_sum`:

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

    >>> my_sum([10, 20, 30])
    60
    """
    # ACCUMULATOR sum_so_far: keep track of the running su
    # of the elements in numbers.
    sum_so_far = 0

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far


def my_sum_v2(numbers: list[int]) -> int:
    """Return the sum of the given numbers.

    >>> my_sum_v2([10, 20, 30])
    60
    """
    # ACCUMULATOR sum_so_far: keep track of the running su
    # of the elements in numbers.
    sum_so_far = 0

    for i in range(0, len(numbers)):
        sum_so_far = sum_so_far + numbers[i]

    return sum_so_far
```

Both `my_sum` and `my_sum_v2` use the accumulator pattern, and in fact initialize and update their accumulator in the exact same way. But there are some key differences in how their loops are structured:

- Loop variable `number` vs. `i`: `number` refers to an element of the list `numbers` (starting with the first element); `i` refers to an index (starting at `0`).
- Looping over a list vs. a `range`: `for number in numbers` causes the loop body to execute once for each element in `numbers`. `for i in range(0, len(numbers))` causes the loop body to execute once for each integer in `range(0, len(numbers))`.[1]
- Updating the accumulator: since `number` refers to a list element, we can add it directly to the accumulator. Since `i` refers to *where* we are in the list, we access the corresponding list element using list indexing to add it to the accumulator.

[1] Because the range "stop" argument is exclusive, these two versions both cause the same number of iterations, equal to the number of elements in `numbers`.

In the case of `my_sum`, both our element-based and index-based implementations are correct. However, our next example illustrates a situation where the loop *must* know the index of the current element in order to solve the given problem.

When location matters

Consider the following problem: given a string, count the number of times in the string two adjacent characters are equal. For example, the string `'look'` has two adjacent `'o'`s, and the string `'David'` has no repeated adjacent characters. The location of the characters matters: even though the string `'canal'` has two `'a'` characters, they are not adjacent.

Let's use these examples to design our function:

```
def count_adjacent_repeats(string: str) -> int:
    """Return the number of times in the given string that two adjacent characters are equal.

    >>> count_adjacent_repeats('look')
    1
    >>> count_adjacent_repeats('David')
    0
    >>> count_adjacent_repeats('canal')
    0
    """
```

Before we try to implement this function, let's reason about how we might approach the problem. First, as this is a "counting" problem, a natural fit would be to use an accumulator variable `repeats_so_far` that starts at 0 and increases by 1 every time two adjacent repeated characters are found. We don't know where the characters in the string may be repeated, so we must start at the beginning and continue to the end. In addition, we are comparing adjacent characters, so we need two indices every loop iteration:

Comparison
`string[0] == string[1]`
`string[1] == string[2]`
`string[2] == string[3]`

…

Notice that the indices to the left of the `==` operator start at 0 and increase by 1. Similarly, the indices to the right of the `==` operator start at 1 and increase by 1. Does this mean we need to use two for loops and two `range`s? No. We should also notice that the index to the right of `==` is always larger than the left by 1, so we have a way of calculating the right index from the left index. Here is out first attempt.

```
def count_adjacent_repeats(string: str) -> int:
    """Return the number of repeated adjacent characters in string.

    >>> count_adjacent_repeats('look')
    1
    >>> count_adjacent_repeats('David')
    0
    >>> count_adjacent_repeats('canal')
    0
    """
    # ACCUMULATOR repeats_so_far: keep track of the number of adjacent
    # characters that are identical
    repeats_so_far = 0

    for i in range(0, len(string)):
        if string[i] == string[i + 1]:
            repeats_so_far = repeats_so_far + 1

    return repeats_so_far
```

Unfortunately, if we attempt to run our doctest examples above, we don't get the expected values. Instead, we get 3 `IndexError`s, one for each example. Here is the error for the first failed example:

```
Failed example:
    count_adjacent_repeats('look')
Exception raised:
    Traceback (most recent call last):
      File "path\to\Python\Python310\lib\doctest.py", line 1350, in __run
        exec(compile(example.source, filename, "single",
      File "<doctest __main__.count_adjacent_repeats[0]>", line 1, in <module>
        count_adjacent_repeats('look')
      File "path/to/functions.py", line 74, in count_adjacent_repeats
        if string[i] == string[i + 1]:
    IndexError: string index out of range
```

Conveniently, the error tells us what the problem is: `string index out of range`). It even tells us the line where the error occurs: `if string[i] == string[i + 1]:`. It is now our job to figure out why the line is causing an `IndexError`. The if condition indexes into the parameter `string` using `i` and `i + 1`, so one of those indexes must be causing the error.

Remember that given a string of length `n`, the valid indices are from `0` to `n - 1`. Now let's look at our use of `range`: `for i in range(0, len(string))`. This means that `i` can take on the values `0` to `n - 1`, which seems to be in the correct bounds. But don't forget, we also are indexing using `i + 1`! This is the problem: `i + 1` can take on the values `1` to `n`, and `n` is not a valid index.

We can solve this bug by remembering our goal: to compare adjacent pairs of characters. For a string of length `n`, the last pair of characters is `(string[n - 2], string[n - 1])`, so our loop variable `i` only needs to go up to `n - 2`, not `n - 1`. Let's make this change:

```
def count_adjacent_repeats(string: str) -> int:
    """Return the number of repeated adjacent characters in string.

    >>> count_adjacent_repeats('look')
    1
    >>> count_adjacent_repeats('David')
    0
    >>> count_adjacent_repeats('canal')
    0
    """
    # ACCUMULATOR repeats_so_far: keep track of the number of adjacent
    # characters that are identical
    repeats_so_far = 0

    for i in range(0, len(string) - 1):
        if string[i] == string[i + 1]:
            repeats_so_far = repeats_so_far + 1

    return repeats_so_far
```

Notice that we could not have implemented this function using an element-based for loop. Having `for char in string` would let us access the current character (`char`), but *not* the next character adjacent to `char`. To summarize, when we want to write a loop body that compares the current element with another based on their positions, we must use an index-based loop to keep track of the current index in the loop.

Two lists, one loop

Index-based for loops can also be used to iterate over two collections in parallel using a single for loop. Consider the common mathematical problem: sum of products.[2]

[2] In your linear algebra course you'll learn about the *inner product* (or *dot product*) operation, which formalizes this idea.

For example, suppose we have two nickels, four dimes, and three quarters in our pocket. How much money do we have in total? To solve this, we must know the value of nickels, dimes, and quarters. Then we can use sum of products:

```
>>> money_so_far = 0.0
>>> money_so_far = money_so_far + 2 * 0.05  # 2 nickels
>>> money_so_far = money_so_far + 4 * 0.10  # 4 dimes
>>> money_so_far = money_so_far + 3 * 0.25  # 3 quarters
>>> money_so_far
1.25
```

This looks very similar to our `sum_so_far` exploration from earlier. The main difference is that this time we are accumulating products using the `*` operator. To the left of the `*` operator, we have a count (e.g., the number of nickels, an `int`). To the right of the `*` operator, we have a cent value (e.g., how much a nickel is worth in cents, a `float`). We can store this information in two same-sized lists. Let's design a function that uses these two lists to tell us how much money we have:

```
def count_money(counts: list[int], denoms: list[float]) -> float:
    """Return the total amount of money for the given coin counts and denominations.

    counts stores the number of coins of each type, and denominations stores the
    value of each coin type. Each element in counts corresponds to the element at
    the same index in denoms.

    Preconditions:
      - len(counts) == len(denoms)

    >>> count_money([2, 4, 3], [0.05, 0.10, 0.25])
    1.25
    """
```

Before using a loop, let's investigate how we would implement this using a comprehension. We need to multiply each corresponding element of `counts` and `denoms`, and add the results:

```
(counts[0] * denoms[0]) + (counts[1] * denoms[1]) + (counts[2] * denoms[2]) + ...
```

We can generate each of these products by using `range`:[3]

[3] We used `len(counts)`, but could have used `len(denoms)` as well because of the function's precondition.

```
[counts[i] * denoms[i] for i in range(0, len(counts))]
```

And we can then compute the sum of this expression by using the built-in Python function `sum`:

```
def count_money(counts: list[int], denoms: list[float]) -> float:
    """Return the total amount of money for the given coin counts and denominations.

    counts stores the number of coins of each type, and denominations stores the
    value of each coin type. Each element in counts corresponds to the element at
    the same index in denoms.

    Preconditions:
      - len(counts) == len(denoms)

    >>> count_money([2, 4, 3], [0.05, 0.10, 0.25])
    1.25
    """
    return sum([counts[i] * denoms[i] for i in range(0, len(counts))])
```

This implementation of `count_money` has all the necessary ingredients that would appear in an equivalent for loop. Here is our alternate implementation of `count_money` using a for loop, but the same structure as `my_sum` from 5.4 Repeated Execution: For Loops.

```
def count_money(counts: list[int], values: list[float]) -> float:
    """...
    """
    # ACCUMULATOR money_so_far: keep track of the total money so far.
    money_so_far = 0.0

    for i in range(0, len(counts)):
        money_so_far = money_so_far + counts[i] * values[i]

    return money_so_far
```

Choosing the right for loop

We have seen two forms of for loops. The first version, the element-based for loop, takes the form `for <loop_variable> in <collection>`. This is useful when we want to process each element in the collection without knowing about its position in the collection. The second version, the index-based for loops, takes the form `for <loop_variable> in <range>`. In index-based for loops, the `range` must belong to the set of valid indices for the collection we wish to loop over. We have seen two situations where this is useful:[4]

[4] We'll see one more example use of index-based loops later this chapter.

1. When the location of elements in the collection matters (as in `count_adjacent_repeats`).
2. When we want to loop through more than one list at a time (as in `count_money`), using the same index for both lists.

You might have noticed from our `my_sum` example that index-based for loops are *more powerful* than element-based for loops: given the current index, we can always access the current collection element, but not vice versa. So why don't we just always use index-based for loops? Two reasons: first, not all collections can be indexed (think `set` and `dict`); and second, index-based for loops introduce a level of *indirection* to our code. In our `my_sum_v2` example, we had to access the current element using list indexing (`numbers[i]`), while in `my_sum`, we could directly access the element by using the loop variable (`number`). So it's important to understand when we can use element-based for loops vs. index-based for loops, as the former makes our code easier to write and understand.

References

- CSC108 videos: For loops over indices (Part 1 only)
- CSC108 videos: Parallel Lists and Strings (Part 1, Part 2)