Topic 4.2: Iteration – The `for` Loop

In the previous chapter, you learned about the `while` loop, which repeats a block of code as long as a condition is `True`. However, many programming tasks involve working through a known collection of items – every character in a string, every element in a list, or every number in a range. For these situations, the cleaner tool is the `for` loop.

Why are there two different loop?

Even though any loop can be implemented with either a `while` loop or a `for` loop, many languages include both. The reason is that there are times when one or the other are easier to write and read. Consider the following code:

```
numbers = ["one", "two", "three"]
index = 0
while index < len(numbers):
    print(numbers[index])
    index += 1
```

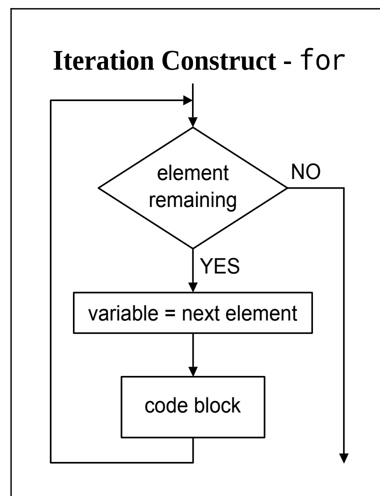
```
numbers = ["one", "two", "three"]
for number in numbers:
    print(number)
```

The `for` version is shorter, easier to read, and eliminates two common mistakes: forgetting to initialize the index, and forgetting to increment it. The `for` loop is designed specifically for stepping through a sequence, one element at a time.

The Syntax of the `for` Loop

```
for variable in sequence:
    # indented block
    statement1
    statement2
```

- The line starts with the keyword `for`.
- Next a loop variable that will hold each element in turn.
- Then the keyword `in`.
- Then any iterable object (list, string, range, tuple, etc.).
- The line ends with a colon (:).
- The indented block runs once for each item in the sequence; each iteration, the loop variable is assigned the next value.



The `range` Function

The `range` function generates a sequence of integers. It's the most common way to run a for loop a specific number of times. The range function takes a variable number of integer parameters and returns the following:

`range(stop)`	from `0` to `(stop-1)`
`range(start, stop)`	from `start` to `(stop-1)`
`range(start, stop, step)`	from `start`, adding `step` each time, until `(stop-1)`

Examples:

Python Code	Resulting Output
`for i in range(5):` `    print(i, end=" ")`	`0 1 2 3 4`
`for i in range(1, 6):` `    print(i, end=" ")`	`1 2 3 4 5`
`for i in range(1, 10, 2):` `    print(i, end=" ")`	`1 3 5 7 9`
`for i in range(5, 0, -1):` `    print(i, end=" ")`	`5 4 3 2 1`

Notice that the `range` function can generate a decreasing sequence, but as with list splicing, the larger value must come before the smaller value for the endpoints. Also notice that, again like list splicing, when generating a decreasing sequence, the lower range value is excluded from the range, not the upper range value.

Unlike when you create a list or tuple, the `range` function does not allocate memory for each element that is to be iterated over. The `range` function returns an object of type `range` that is able to calculate the each next value for iteration. This is a much more space-efficient way to generate values to iterate over. If a list or tuple is required from a range, this can be done by calling the list or tuple constructor.

```
>>> print(range(25))
range(0, 25)

>>> sys.getsizeof(range(25))
48

>>> print(tuple(range(25)))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24)

>>> sys.getsizeof(tuple(range(25)))
240
```

When an Index is Required: The `enumerate` Function

Sometimes you need both the element and its position. enumerate() returns pairs of (index, element):

```
fruits = ["apple", "banana", "cherry"]

for index, fruit in enumerate(fruits):
    print(index, ":", fruit)
```

```
0 : apple
1 : banana
2 : cherry
```

This is much cleaner than using a while loop with a manual index variable.

Exercise