# 11. Loops

## The `for` Loop

The `for` loop is used to iterate over a sequence (such as a list, tuple, or string) or other iterable objects. Its basic syntax is:

```
for element in iterable:
    # code block
    # statements
```

Here's how the `for` loop works:

1. The loop starts by retrieving the first item from the `iterable` (e.g., a list).
2. The current item is assigned to the `element` variable.
3. The code block under the `for` loop is executed, using the current value of `element`.
4. The loop moves on to the next item in the `iterable` and repeats steps 2 and 3.
5. This process continues until all items in the `iterable` have been processed.

Example:

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

for fruit in fruits:
    print(f"I like {fruit}s.")
```

Output:

```
I like apples.
I like bananas.
I like cherries.
```

## The `range()` Function

The `range()` function is often used with the `for` loop to generate a sequence of numbers. It takes one, two, or three arguments:

- `range(stop)`: generates a sequence of numbers from 0 up to (but not including) the `stop` value.
- `range(start, stop)`: generates a sequence of numbers from `start` up to (but not including) `stop`.
- `range(start, stop, step)`: generates a sequence of numbers from `start` up to (but not including) `stop`, incrementing by `step` each time.

Example:

```
for i in range(5):
    print(i)
```

Output:

```
0
1
2
3
4
```

# The `while` Loop

The `while` loop is used to repeatedly execute a block of code as long as a given condition is true. Its basic syntax is:

```
while condition:
    # code block
    # statements
```

Here's how the `while` loop works:

1. The `condition` is evaluated. If it is `True`, the code block under the `while` loop is executed.
2. After executing the code block, the `condition` is evaluated again.
3. If the `condition` is still `True`, the code block is executed again.
4. This process continues until the `condition` becomes `False`.

Example:

```
count = 0
while count < 5:
    print(f"Count is {count}")
    count += 1
```

Output:

```
Count is 0
Count is 1
Count is 2
Count is 3
Count is 4
```

# Loop Control Statements

Python provides additional statements to control the flow of loops:

- `break`: terminates the current loop and transfers execution to the statement immediately following the loop.
- `continue`: skips the remaining code in the current iteration and moves to the next iteration of the loop.

Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Output:

```
0
1
2
3
4
```

```
for i in range(10):
    if i == 5:
```

```
        continue
    print(i)
```

Output:

```
0
1
2
3
4
6
7
8
9
```

# Nested Loops

Loops can be nested inside other loops. This means that for each iteration of the outer loop, the inner loop is executed completely. Nested loops are useful when working with multi-dimensional data structures, such as lists of lists or matrices.

Example:

```
numbers = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for outer_list in numbers:
    for inner_value in outer_list:
        print(inner_value)
```

Output:

```
1
2
3
4
5
6
7
```

```
8
9
```

See Also

12. Turtle Module (Intro)