◈ ◈While loop◈ ◈

Python utilizes the while loop similarly to other popular languages. The while loop evaluates a condition then executes a block of code if the condition is true. The block of code executes repeatedly until the condition becomes false.

OR

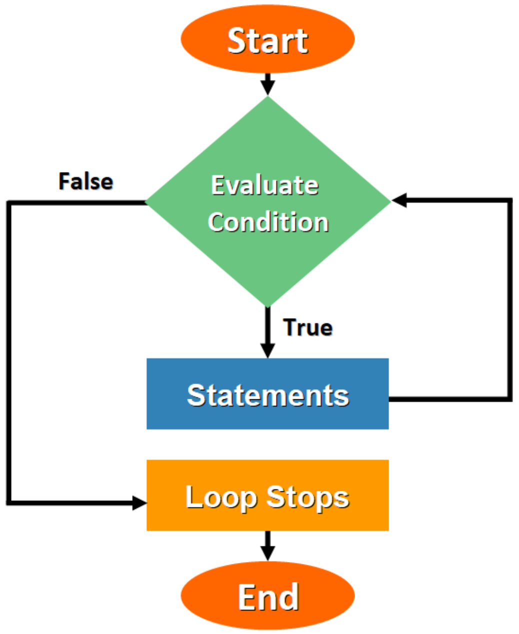
They are used to repeat a sequence of statements an unknown number of times. This type of loop runs **while** a given condition is True and it only stops when the condition becomes False.

When we write a while loop, we don't explicitly define how many iterations will be completed, we only write the condition that has to be True to continue the process and False to stop it.

◈ How While Loops Work

In [1]:
```
from IPython.display import Image
Image(filename='C:\\Users\\deepali\\OneDrive\\Desktop\\while loops notes.png', width=500)
```

Out[1]:



- The process starts when a while loop is found during the execution of the program.
- The condition is evaluated to check if it's True or False.
- If the condition is True, the statements that belong to the loop are executed.
- The while loop condition is checked again.
- If the condition evaluates to True again, the sequence of statements runs again and the process is repeated.
- When the condition evaluates to False, the loop stops and the program continues beyond the loop.

The basic syntax:

```
while expression:
    statement(s)
```

- <statement(s)> represents the block to be repeatedly executed, often referred to as the body of the loop. This is denoted with indentation, just as in an if statement.
- The controlling expression, **expression**,typically involves one or more variables that are initialized prior to starting the loop and then modified somewhere in the loop body.
- When a while loop is encountered, **expression** is first evaluated in **Boolean context**. If it is true, the loop body is executed.Then **expression** is checked again, and if still true, the body is executed again. This continues until **expression** becomes false, at which point program execution proceeds to the first statement beyond the loop body.

In [2]:
```
n = 5
while n>0:
    n-=1
    print(n)
```

```
4
3
2
1
0
```

Here’s what’s happening in this example:

- n is initially 5. The expression in the while statement header on line 2 is n > 0, which is true, so the loop body executes. Inside the loop body on line 3, n is decremented by 1 to 4, and then printed.
- When the body of the loop has finished, program execution returns to the top of the loop at line 2, and the expression is evaluated again. It is still true, so the body executes again, and 3 is printed.
- This continues until n becomes 0. At that point, when the expression is tested, it is false, and the loop terminates. Execution would resume at the first statement following the loop body, but there isn't one in this case.

Note that the controlling expression of the while loop is tested first, before anything else happens. If it's false to start with, the loop body will never be executed at all:

In [3]:
```python
n = 0
while n>0:
    n-=1
    print(n)
```

In the example above, when the loop is encountered, n is 0. The controlling expression n > 0 is already false, so the loop body never executes.

Example

In [4]:
```python
days = 0
week = ['Monday','Tuesday','Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']

while days<7:
    print("Today is ",week[days])
    days+=1
```

```
Today is  Monday
Today is  Tuesday
Today is  Wednesday
Today is  Thursday
Today is  Friday
Today is  Saturday
Today is  Sunday
```

Line-by-Line Explanation of the above CODE:

1. the variable ‘days’ is set to a value 0.
2. a variable week is assigned to a list containing all the days of the week.
3. while loop starts
4. the block of code will be executed until the condition returns ‘true’.
5. the condition is ‘days<7’ which rougly says run the while loop till the point the variable days is less than 7
6. So when the days=7 the while loop stops executing.
7. the days variable gets updated on every iteration.
8. When the while loop runs for the first time the line ‘Today is Monday’ is printed onto the console and the variable days becomes equal to 1.
9. Since the variable days is equal to 1 which is less than 7 so the while loop is executed again.
10. It goes on again and again and when the console prints ‘Today is Sunday’ the variable days is now equal to 7 and the while loop stops executing.

Here’s another while loop involving a list, rather than a numeric comparison:

In [5]:
```python
L = [ " TCS "," Infosys" , " Accenture"]

while L :
    print(L.pop(-1))
```

```
Accenture
Infosys
TCS
```

When a list is evaluated in Boolean context, it is truthy if it has elements in it and falsy if it is empty. In this example, L is true as long as it has elements in it. Once all the items have been removed with the .pop() method and the list is empty, L is false, and the loop terminates.

What are Infinite While Loops?

Infinite loops are typically the result of a bug, but they can also be caused intentionally when we want to repeat a sequence of statements indefinitely until a break statement is found.

In [7]:

```python
# Define a variable
i = 5

# Run this loop while i is less than 15
while i < 15:
    # Print a message
    print("Hello, World!")
```

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

The value of the variable i is never updated (it's always 5). Therefore, the condition i < 15 is always True and the loop never stops.

If we run this code, the output will be an "infinite" sequence of Hello, World! messages because the body of the loop print("Hello, World!") will run indefinitely.

To stop the program, we will need to interrupt the loop manually by pressing CTRL + C.

To fix this loop, we will need to update the value of i in the body of the loop to make sure that the condition i < 15 will eventually evaluate to False.

This is one possible solution, incrementing the value of i by 2 on every iteration:

In [8]:
```python
i = 5

while i < 15:
    print("Hello, World!")
    # Update the value of i
    i += 2
```

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

You just need to write code to guarantee that the condition will eventually evaluate to False.

◈ The Python "break" and "continue" Statements

Python provides two keywords that terminate a loop iteration prematurely:

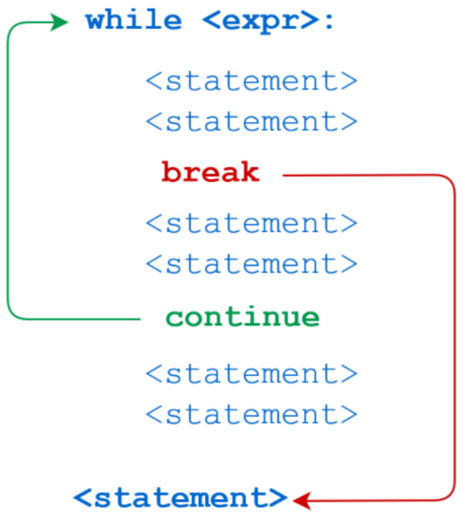
- The Python **break** statement immediately terminates a loop entirely. Program execution proceeds to the first statement following the loop body.
- The Python **continue** statement immediately terminates the current loop iteration. Execution jumps to the top of the loop, and the controlling expression is re-evaluated to determine whether the loop will execute again or terminate.

The distinction between **break and continue is** demonstrated *in* the following diagram:

In [9]:
```python
from IPython.display import Image
Image(filename='C:\\Users\\deepali\\OneDrive\\Pictures\\Screenshots\\break_continue.png', width=300)

```

Out[9]:



break and continue

Here’s a script file called break.py that demonstrates the break statement:

In [10]:

```
1  n = 5
2  
3  while n>0:
4      n-=1
5      if n == 2:
6          break
7      print(n)
8  print(" Loop Ended!")
```

```
4
3
 Loop Ended!
```

When n becomes 2, the break statement is executed. The loop is terminated completely, and program execution jumps to the print() statement on line 7.

The next script, continue.py, is identical except for a continue statement in place of the break:

In [11]:

```
1  n = 5
2  
3  while n > 0:
4      n-=1
5      if n ==2 :
6          continue
7      print(n)
8  print("Loop Ended!")
```

```
4
3
1
0
Loop Ended!
```

This time, when n is 2, the continue statement causes termination of that iteration. Thus, 2 isn't printed. Execution returns to the top of the loop, the condition is re-evaluated, and it is still true. The loop resumes, terminating when n becomes 0, as previously.

In [ ]:
```

```

In [ ]:
```

```

◈ The "else" Clause

```
while <expr>:
    <statement(s)>
else:
    <additional_statement(s)>
```

```
while <expr>:
    <statement(s)>
<additional_statement(s)>
```

What’s the difference?
In the latter case, without the else clause, <additional_statement(s)> will be executed after the while loop terminates, no matter what.

When <additional_statement(s)> are placed in an else clause, they will be executed only if the loop terminates “by exhaustion”—that is, if the loop iterates until the controlling condition becomes false. If the loop is exited by a break statement, the else clause won’t be executed.

Example

In [12]:
```python
n = 5
while n> 0:
    n-=1
    print(n)
else:
    print("Loop Ended!")
```

```
4
3
2
1
0
Loop Ended!
```

In this case, the loop repeated until the condition was exhausted: n became 0, so n > 0 became false. Because the loop lived out its natural life, so to speak, the else clause was executed. Now observe the difference here:

In [13]:
```python
n = 5
while n > 0:
    n-=1
    print(n)
    if n==2:
        break

else:
    print("Loop Ended!")
```

```
4
3
2
```

◈ User Input Using a While Loop

We will the input() function to ask the user to enter an integer and that integer will only be appended to list if it's even.

In [14]:
```
# define a list

nums_list = []

# the loop will run while the len of the (num_list) is less than 4

while len(nums_list) < 4:

        # Ask for user input and store it in a variable as an integer.

    user_input = int(input("Enter an integer : "))
        # If the input is an even number, add it to the list
    if user_input % 2==0:
        nums_list.append(user_input)
```

```
Enter an integer : 1
Enter an integer : 2
Enter an integer : 3
Enter an integer : 4
Enter an integer : 5
Enter an integer : 6
Enter an integer : 7
Enter an integer : 8
```

In [15]:
```python
nums = []

# The loop will run while the length of the
# list nums is less than 4
while len(nums) < 4:
    # Ask for user input and store it in a variable as an integer.
    user_input = int(input("Enter an integer: "))
    # If the input is an even number, add it to the list
    if user_input % 2 == 0:
        nums.append(user_input)
```

```
Enter an integer: 1
Enter an integer: 2
Enter an integer: 5
Enter an integer: 4
Enter an integer: 7
Enter an integer: 8
Enter an integer: 6
```

In [16]:
```python
nums
```

Out[16]:
```
[2, 4, 8, 6]
```

In [17]:
```python
nums_list
```

Out[17]:
```
[2, 4, 6, 8]
```

Let's analyze this program line by line:

1. We start by defining an empty list and assigning it to a variable called nums.
2. Then, we define a while loop that will run while len(nums) < 4.
3. We ask for user input with the input() function and store it in the user_input variable.
   💡 Tip: We need to convert (cast) the value entered by the user to an integer using the int() function before assigning it to the variable because the input() function returns a string (source).
4. We check if this value is even or odd.
5. If it's even, we append it to the nums list.
6. Else, if it's odd, the loop starts again and the condition is checked to determine if the loop should continue or not.
   If we run this code with custom user input, we get the following output:

```
Enter an integer: 1
Enter an integer: 2
Enter an integer: 3
Enter an integer: 4
Enter an integer: 5
Enter an integer: 6
Enter an integer: 7
Enter an integer: 8
```

7. Before a "ninth" iteration starts, the condition is checked again but now it evaluates to False because the nums list has four elements (length 4), so the loop stops.
   If we check the value of the nums list when the process has been completed, we see this:

```
nums
[2, 4, 6, 8]
```

These are some examples of real use cases of while loops:

- User Input: When we ask for user input, we need to check if the value entered is valid. We can't possibly know in advance how many times the user will enter an invalid input before the program can continue. Therefore, a while loop would be perfect for this scenario.
- Search: searching for an element in a data structure is another perfect use case for a while loop because we can't know in advance how many iterations will be needed to find the target value. For example, the Binary Search algorithm can be implemented using a while loop.
- Games: In a game, a while loop could be used to keep the main logic of the game running until the player loses or the game ends. We can't know in advance when this will happen, so this is another perfect scenario for a while loop.

◈ In Summary

- While loops are programming structures used to repeat a sequence of statements while a condition is True. They stop when the condition evaluates to False.
- When you write a while loop, you need to make the necessary updates in your code to make sure that the loop will eventually stop.
- An infinite loop is a loop that runs indefinitely and it only stops with external intervention or when a break statement is found.
- You can stop an infinite loop with CTRL + C.
- You can generate an infinite loop intentionally with while True.

- The break statement can be used to stop a while loop immediately.