TECHNION

Azrieli Continuing Education and External Studies Division

Module 5.4.1: loops and iteration

Wait a minute..

Let’s get to know George.

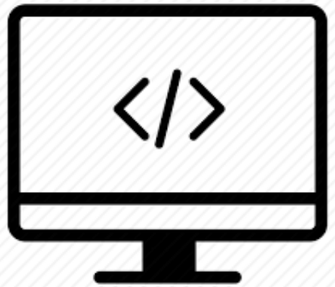
Meanwhile in class..

Poor George..

Reality

George has to write down a hundred times “I’m Sorry”

Programming



What if we had to write the same thing a hundred times in Python?

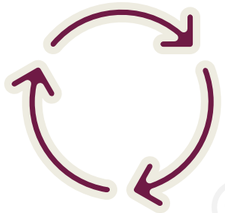
Loops

- Loops help us whenever we wish to run a block of code multiple times.

George has to write down a hundred times “I’m Sorry”

print (“I’m Sorry”)

x100



While Loop

- A *while* loop in Python, helps us repeat a group of statements as long as a condition is **true**!
- We can look at the '*while*' expression as – "*As long as*.."
- It requires a **condition** and **statements**.
- A *while* loop receives a *Boolean condition*.
- *As long as a* condition evaluates as *True*, the code block inside a '*while'* loop executes again and again.
- We create a *while* loop in Python using the '*while'* key-word!

While Loop – Syntax

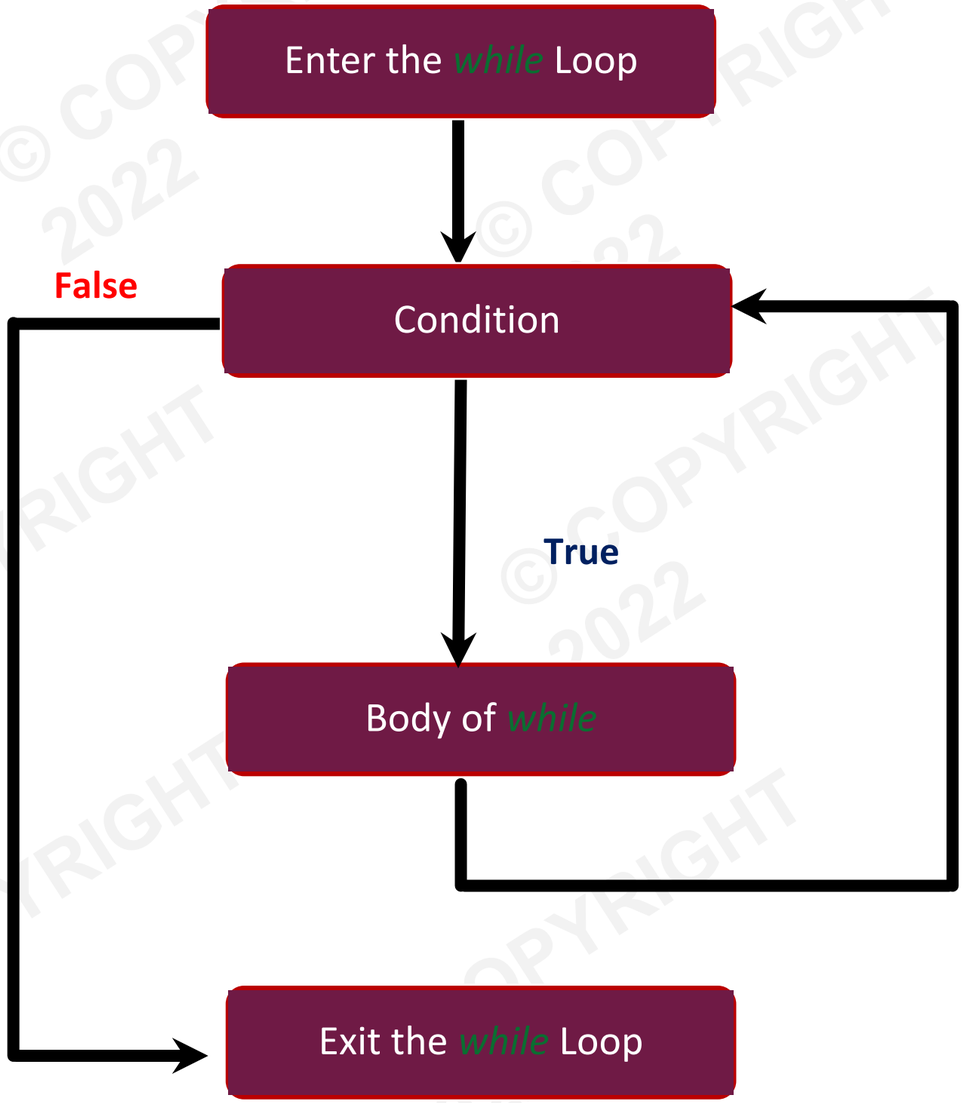
while (*Boolean Condition*):
Statements

- Starts with '*while*' keyword.
- A *Boolean Condition*.
- A colon. (**:**)
- The line after the colon **must** be indented. (4 Spaces)
- Indented Statements.

While Loop – Flow Chart

- ***As long as*** the Boolean Condition evaluates as *True*, the *while* loop keeps on looping!

- ***Once*** the *Boolean Condition* evaluates as *False*, we exit the *while* loop!



While Loop – Example

```
num = 0
while (num < 10):
        print(num)
        num = num + 1
```

- Create an a variable and name it as "**num**" *before-hand*!
- Declare a *while* loop using the '*while*' keyword!
- ***As long as*** the *Boolean Condition* evaluates as true:
  Print out the value of **num**.
  Update the value of **num** by assigning the value of **num** + 1 to **num**
- ***Once*** the *Boolean Condition* evaluates as ***false***, we *exit* the loop!

Infinite Loops

- What is wrong with this loop?

```
num = 5
while (num > 0):
        print(num)
```

Useless Loops

- What is wrong with this loop?

```
num = 0
while (num > 10):
        print(num)
```

For Loop!

- A *for* loop in Python, helps us repeat a group of statements a ***specified*** number of times!

- It requires *three* features in order to work.
- → *(Iterating Variable, Statements, Sequence)*

- We declare a *for* loop in Python using the '*for*' key-word!

The Three Features

- In order to use a *for* Loop in Python, it has to consist ***three*** features:

Iterating Variable

An indicator
For Starting State

Statement(s)

What to do next

Sequence

Something to Loop over

Iterating Variable

- An *iteration* is one step in a loop.

- George had to write down a hundred times that he is sorry.

→ Once George writes down for the first time ”I’m Sorry”,he makes an iteration of the punishment.

print (“I’m Sorry”)

- An *Iterating Variable* is a variable used as an indicator for the
- program to keep track on our loop iterations!

Statement

- *Statement(s)* is a block of code indented inside a *for* loop.

- For each iteration of a loop, the desired block of code is being executed!

- Without mentioning any statement, we will face a *Syntax Error*!

Sequence

- A *sequence* is a collection of some objects.
- A *sequence* can be anything we can iterate over!
- For instance →
- - A string is a collection of characters that we can loop over it, character by character!
- - A simple integer/float *does not* have an iterator.

For Loop – Syntax

for iterating_var in sequence:
Statement(s)

- Starts with '*for*' keyword.
- A *Boolean Condition*.
- A colon. (**:**)
- The line after the colon **must** be indented. (4 Spaces)
- Indented Statement.

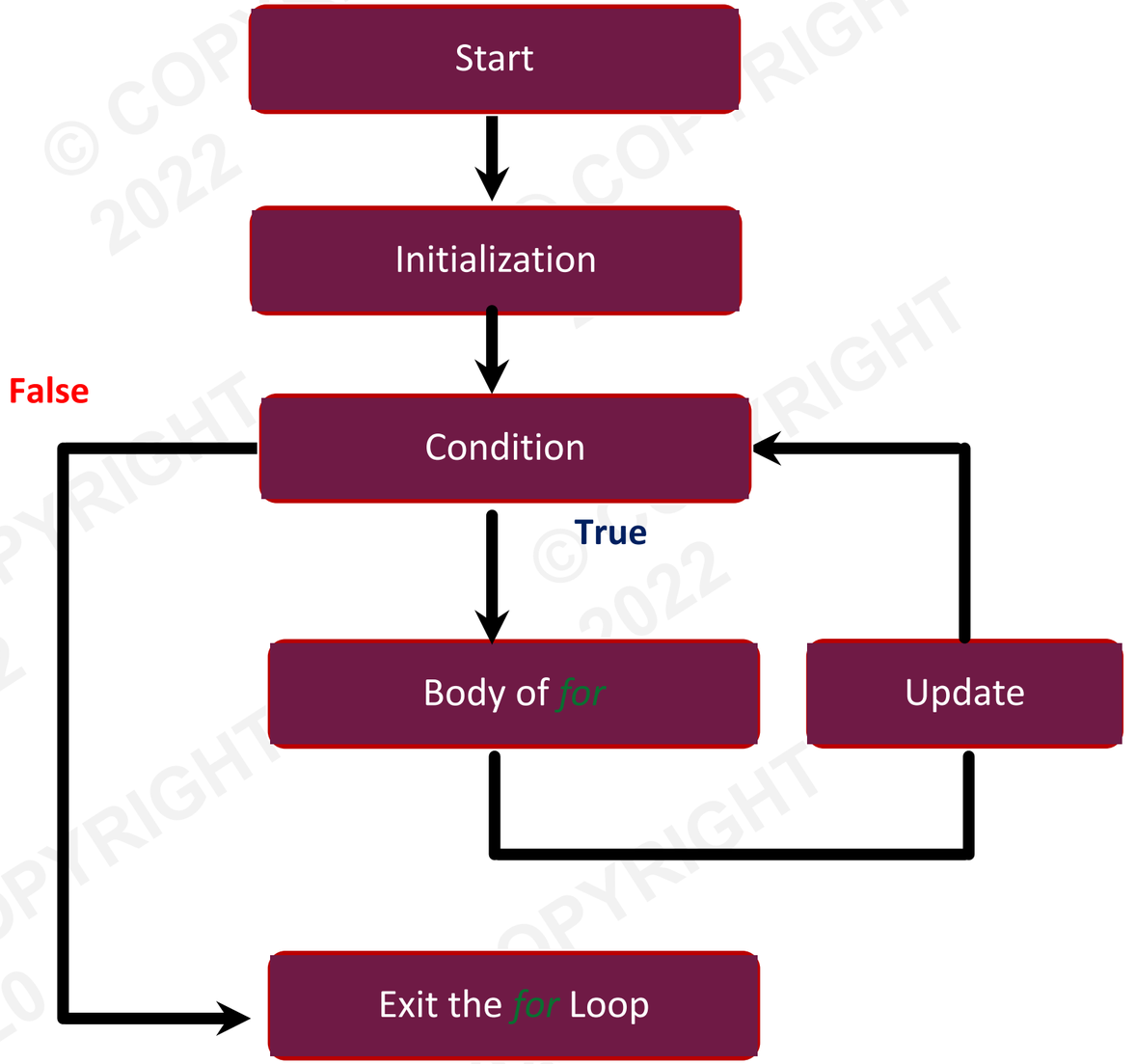
Implementation of the three-features

```
for letter in 'CYBER':
    print(number)
```

- *Iterating Variable* → letter
- *Sequence* → 'CYBER'
- Statement → *print*(letter)

For Loop – Flow Chart

- *Initialization* is the part where we create the loop.
- *Initialization* is also the part where we assign the ***first*** value into the *iterating variable*.
- ***Once*** the *Boolean Condition* evaluates as *True*, we enter the body of the *for* loop and execute the *statements*!
- ***Once*** the first iteration is done, we update the *iterating variable* and check the *Boolean Condition* once again!
- ***Once*** the *Boolean Condition* evaluates as *False*, we exit the *for* loop!



Looping Over Numbers

- Many times, we would like to loop over numbers, in order.
- In the previous example, we created a *sequence*, using characters of a string.
- What if we want to create a *sequence* of numbers?
- Let's get to know the *range*() function!

range() Function!

- Using the *range()* function, we can create a *sequence* of numbers.
- For instance →
- *range(10)* creates a *sequence* of numbers between *0* and *10.*
-
- Counting in Python always *starts with 0*, and *ends on **n**-1*, with ***n*** being the length of the object being counted.

- *range(10)* →
- *(0,10)* →
- *[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]* →
- *starts with 0, ends with 10-1*

range() Function – List View

- We can look on a *range*() as a *list*!
- Calling out *range*(10) returns as:
- *range*(0, 10)

- We can use the *list*() function to cast a range *sequence* into a *list*!

```
list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

For Loop – Example

```
for number in range(10):
    print(number)
```

- Declare a for loop using the '*for*' keyword!
- Create an *Iterating Variable* "**number**"!
- Create a *sequence* of digits from 0 to 9, using *range(10)*.
- In the *first iteration*, the *first* value in the *sequence* is assigned to *number*. (0)
- *For each iteration* of the *sequence*, we execute the statement: print out the value of stored inside *number*.
- The value stored inside *number* is set to the next value stored inside the *sequence*!

Looping Over Numbers - Demonstration

```
for number in range(10):
    print(number)
```

```
for number in [0,1,2,3,4,5,6,7,8,9]:
    print(number)
```

Same Output!

Looping Over Numbers – George!

George has to write down a hundred times “I’m Sorry”

```
for number in range(100):
    print(number, “I’m Sorry!”)
```

range() Function - Arguments

- Instead of only specifying the *upper limit* of the *range* we can also specify the *bottom limit*:
- *range*(a, b)

- The *sequence* is going to start from **a** to **b-1**.
- For Instance:
- *range*(3,10) → Creates a *sequence* of digits starting from 3 and ends on 9.
- We can also add a *third argument*, to specify the exact number to *increment/decrement* by:

range(a, b, c)

For instance:

range(2, 11, 2) → Creates a sequence of digits starting from 2 and ends on 10. ([2, 4, 6, 8, 10])

For Loop Variable Names

- *Iterating Variables* i, j, and k are the conventional variable names used for loops.

- That being said, if the iterations have meaning, a more indicative name should be used, to improve readability:

```
for celebrity_name in [‘The Rock’, ‘Bruno Mars’, ’Cristiano Ronaldo’]:
        print(celebrity_name)
```

When do we use FOR, and when do we use WHILE?

- Rule of thumb:

- If you know ahead of the loop execution how many loops you will want, use *for*.

- If you will only know when to stop the loop during the loop execution, use *while*.

Nested Loops

- Remember Nested *if* Condition?
  (an *if* statement inside an *if* state

```
if (flavor == "Chocolate"):
    print("Yes!")
    if (price < 15):
        print("Buy")
```

- A loop inside another loop is called

Nested Loops

- Python allows to use one loop inside another loop!
- A Nested loop can be either a Nested '*for*' loop or a Nested '*while*' loop!
- A Nested loop can also be a mix of the two.

Nested for Loop – Syntax

```
for iterating_var in sequence:
        for iterating_var2 in sequence2:
                Statements
        Statements
```

- A colon for both loops. (:)
- The line after the colon **must** be indented. (4 Spaces)
- Indented Statements for each loop!

Nested for Loop – Logic

```
for iterating_var in sequence:
        for iterating_var2 in sequence2:
                Statements
        Statements
```

- Declare a '*for*' loop. (We are going to refer this as "***outer loop"***)
- Declare another '*for*' loop inside the ***outer*** '*for*' loop. (We are going to refer this as "***inner loop"***)
- For every iteration of the the ***outer*** loop, the ***inner*** loop executes.
- The ***inner*** loop continues until <u>all iterations</u> over the given *sequence* are complete and then heads back to the ***outer*** loop.

Nested for Loop - Example

```
for i in range (1,11):
        for j in range(1,11):
                print ( i * j, end ='/t')
        print()
```

```
1  2  3  4  5  6  7  8  9  10
2  4  6  8  10 12 14 16 18 20
3  6  9  12 15 18 21 24 27 30
4  8  12 16 20 24 28 32 36 40
5  10 15 20 25 30 35 40 45 50
6  12 18 24 30 36 42 48 54 60
7  14 21 28 35 42 49 56 63 70
8  16 24 32 40 48 56 64 72 80
9  18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

- The *outer for* loop assign the value 1 to **i** in the *first* iteration.
- *Once* the *outer* loop statements are being executed, an *inner* loop is being called, assigning the value 1 to **j.**
- In the *first* iteration of the *inner* loop, we *print* the value of **i * j** (**1 * 1** = 1).
  → In the next iteration of the *inner* loop, we assign the next in the inner loop sequence (2) to **j**
  → we *print* again **i** * **j** (**1 * 2** = 2)
- Once the *inner* loop iteration is over, the *outer* loop iterates, and we assign the next value to **i** (2), and the *inner* loop starts all over!

Nested while Loop – Syntax

```
while (Condition):
        while (Condition):
                Statements
        Statements
```

- A colon. (**:**)
- The line after the colon **must** be indented. (4 Spaces)
- Indented Statements.

Nested while Loop – Logic

```
while (Condition):
        while (Condition):
                Statements
        Statements
```

- Declare a '*while*' loop. (*outer loop*)
- Declare another '*while*' loop inside the *outer* '*while*' loop. (*inner loop*)
- *Once* the *outer* loop condition evaluates as *True*, the *inner* '*while*' loop is being called.
- *Once* the *inner* loop condition evaluates as *True*, the *inner* statements are executed.
- *As long as the inner* loop condition evaluates as *True*, the *inner* loop keeps iterating.
- *As long as the outer* loop condition evaluates as *True*, the *outer* loop keeps iterating.

pass

- The *pass* command is a place-holder.
- It continues execution with the next line.

- This is very useful when a *loop* is written, in which an indented code block is *needed* but you want to deal with it *later*.

```
for i in range(100):
    pass
```

break

- The *break* command *stops* the current loop, and jumps *out of it*.

- Example:

```
for i in range(100):
        if (i == 3):
                break
        print (i)
```

continue

- The *continue* command *stops* the current iteration in the loop, *continuing* with the *next* iteration.

- Example:

```
for i in range(10):
        if (i % 3):
                continue
        print (i)
```

The Guessing Game

- Let's create a guessing game, in which a user needs to guess the number 5!

```
number = int(input("Please guess a number!"))
while number != 5:
    print ("Nope, wrong guess!")
    number = int(input("Please guess again!"))
    print ("Nicely Done!, It was 5!")
```

Summary

- Loops
- “*For*” and “*While*” loops
- Iterations
- Nested Loops
- Using Pass, Break and Continue