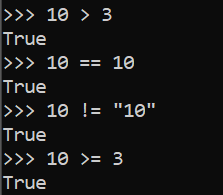


**Comparison Operators**:

We use comparison operators to compare values.

🡨comparing integer values.



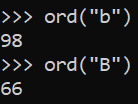
*Why we get True*?

🡪 It is because when we sort these two words, bag comes after.



*Why false*?

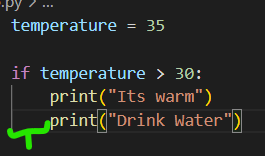
🡪 Because every character here has a numeric representation in programming.

🡨To see a character’s numerical value use ord function.

**Conditional Statements**:

In almost every program, there are times you need to make decisions and that is when you use an if statement.

*After* if *you add a* ***condition*** *which is actually a* ***Boolean expression*** *and if that expression evaluates to* ***True****, the following statements will be* ***executed***.

And **Do not Forget** to terminate the if statement with **:** colon.  


Notice the indentation in the print statements below if condition. *As long as they are indented, they belong to if block*.

temperature = 15

if temperature > 30:

    print("Its warm")

    print("Drink Water")

elif temperature > 20:

    print("Its nice")

else:

    print("Its cold")

print("Done")

The last “done” statement will be executed every time despite conditions above are true or false.

else statement will be executed if none of the above conditions are true.

O/P: Its cold

Done

**Ternary Operator**:

**Example**: We are building an application for university and we want to check if the person who is applying for this university program is eligible or not.

age = 22

if age >= 18:

    print("eligible")

else:

    print("not eligible")

O/P: eligible

There is nothing wrong with this piece of code but there is a cleaner way to achieve the same result.

age = 22

message = "Eligible" if age >= 18 else "Not eligible"

print(message)

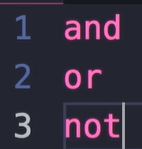
O/P: Eligible

This single statement is almost like plain English and replaces four lines of code.

So what we have here is *Ternary* operator.

**Logical Operators**:

In python we have three logical operators and we use these operators to handle more complex conditions.



**Example**: Imagine building an application for processing loans, so we have two basic conditions that we want to implement.

high\_income = True

good\_credit = True

If both are true then they are eligible for loan.

if high\_income and good\_credit:

    print("Eligible")

If any condition is false,

high\_income = False

good\_credit = True

if high\_income and good\_credit:

    print("Eligible")

else:

    print("Not Eligible")

O/P: Not Eligible

With and operator: if both conditions are true, result = True

With or operator: At least one condition is true, result = True.

not operator inverses the value of a Boolean.

student = True

if not student:

    print("Eligible")

else:

    print("Not Eligible")

O/P: Not Eligible.

if (high\_income or good\_credit) and not student:

We can write complex real world conditions using just these three operators.

**Short Circuit Evaluation**:

high\_income = True

good\_credit = True

student = True

if high\_income and good\_credit and not student:

    print("Eligible")

As per example from last lecture, A person is eligible for loan if they have high income and good credit and not a student.

Now one thing we need to know about these Boolean operators is that they are short circuit.

🡪It means when python interpreter wants to evaluate this expression, it starts from the first argument.

🡪If it is true it continues the evaluation to see if the second argument is also true.

🡪So it continues the evaluation all the way up to end of this expression.

However as soon as one of these arguments is false, the evaluation stops. This is what we call *short circuiting, Just like short circuit concept in electronics*.

**Chaining comparison operators**:

**Example**:

Age should be between 18 and 65.

*First way*:

age = 22

if age >= 18 and age < 65:

    print("Eligible")

*Second and cleaner way*:

age = 22

if 18 <= age < 65:

    print("Eligible")

As you can see second way is very much like how we write an expression in math. This is what we call chaining comparison operators.

**For Loops**:

There are times when we want to repeat a task a number of times.

For example, we send a message to the user, if that message cannot be delivered, perhaps we want to retry 3 times.

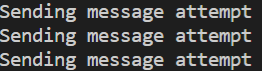
This where we use *loops to create repetition*,

for number in range(3):

    print("Sending message attempt")

Note: range is a built in function, where we pass a value depending on how many times we want to repeat a task.

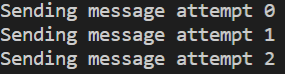
Inside the for block, we mention statements that we want to see repeated.



Let us see what is the *number* that we used in for loop so we will print this as well,

for number in range(3):

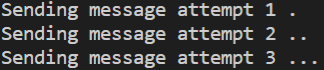
    print(f"Sending message attempt {number}")

🡨This *number* is a variable of type integer. In each iteration we see a different value of the number.

A little more fun with this…

for number in range(3):

    print(f"Sending message attempt {number + 1} {(number + 1) \* '.'}")



A cleaner way to do this is to *add a* ***start – end*** *value in range*, so that we do not have to add one in print statements,

for number in range(1, 4):

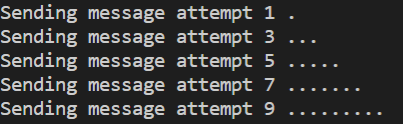
    print(f"Sending message attempt {number} {(number) \* '.'}")

and we get the same result.

Third argument we can add in range is *step*,

for number in range(1, 10, 2):

    print(f"Sending message attempt {number} {(number) \* '.'}")



**For…Else**:

Continuing with example from last lecture, Let us imagine a scenario where after the first attempt we can successfully send a message. In that case we want to jump out of the loop.

for number in range(3):

    print("Sending message attempt")

We do not want to repeat this task of sending message 3 times.

To simulate the scenario of successfully sending a message,

successful = True

If successful after first attempt, we will use break to break out of loop and no more attempts after that.

for number in range(3):

    print("Sending message attempt")

    if successful:

        print("Success")

        break

Let us take this a step further, what if even after 3 attempts, we are still not able to send a message. Perhaps we want to display a different message to the user.

successful = False

for number in range(3):

    print("Sending message attempt")

    if successful:

        print("Success")

        break

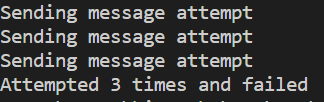
else:

    print("Attempted 3 times and Failed")

We use else at the end of for loop or as we like to call for-else

statement.

“*The code after else block will be executed only when the corresponding for loop completes without early termination*”.



**Nested Loop**:

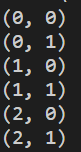
In programming we have a concept called nested loops, so we can put one loop inside another loop and with this we get some interesting results.

for x in range(3):

    for y in range(2):

        print(f"({x}, {y})")

Output:



Here we have two loops, ***x*** *in range* is outer loop and ***y*** *in range* is inner loop.

🡪 Execution of program starts from ***x*** *in range* and x gets the value 0.

🡪Now we get to the next line ***y*** *in range* which is a child of previous for statement. This statement ***y*** *in range* itself is a loop so the print statement that we have inside this loop will be executed 2 times (*range(2)*). Hence, y = 0 and y = 1.

🡪 Now we are done with the inner loop ***y*** *in range* as it has run both its iterations but outer loop ***x*** *in range* has only run its first iteration. So still x=0.

🡪As a result control moves back to outer loop ***x*** *in range* to run its second iteration and then we move to next line ***y*** *in range* again running and completing all its iterations.

“*In this case, control will come out of nested loops only when outer loop has completed all its iterations*”.

**Iterables**:

Let us dive deeper to see what this range function returns.

print(type(range(5)))

O/P:

<class 'range'>

range function *returns an object of type range*.

In python we have primitive types like numbers, strings and Booleans but we also have complex types.

range is one of the examples of those complex types.

The interesting thing about this range object is that it is *iterable*.

Which means we can iterate over it or use it in a for loop.

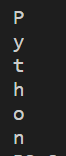
Inside, [*for x in range(5)*:] x will have different value in each iteration.

Now range objects are not the only iterable object in Python.

Strings are also iterable,

for x in "Python":

    print(x)

O/P: 

Another complex types is list, which we use to store a list of objects.

for x in [1, 2, 3]:

    print(x)

O/P: 

**While Loop**:

So we learned that we can use for loop to iterate over Iterable objects. In Python we have another kind of loop that we use to *repeat something as long as the condition is true* which we call as while loop.

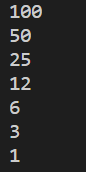
number = 100

while number > 0:

    print(number)

    number //= 2

O/P:



In above example we are not iterating over an iterable like a range object or string or list instead *we are evaluating a condition and repeating a task*.

*Real world example*:

Create an interactive shell where we ask user for input. User should not come out of this shell unless he type “quit”.

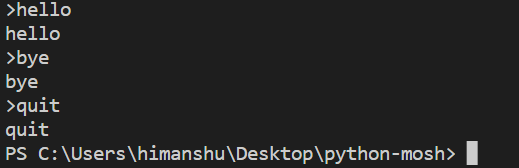
command = ""

while command != "quit":

    command = input(">")

    print(command)

O/P:



Note: An experienced programmer will use this implementation to handle upper, lower case versions of ‘quit’.

command = ""

while command.lower() != "quit":

    command = input(">")

    print(command)

with this, it does not matter how user types the word quit (QUIT/Quit)…it will always terminate the program.

**Infinite Loops**:

If we add True after while statement loop will run forever (since true is always true). So to jump out of this we need a *break* statement.

while True:

    command = input(">")

    print(command)

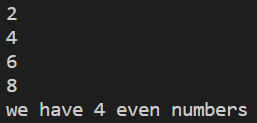
    if command.lower() == "quit":

        break

With this we do not need to initialize command to an empty string. We must always have a way to jump out of infinite loops.

**Exercise**:

We should get this output,



Solution:

count = 0

for number in range(1, 10):

    if number % 2 == 0:

        print(number)

        count += 1

print(f"we have {count} even numbers")