Module 3: Boolean values, Conditional execution, Loops, Lists and List processing, Logical and Bitwise operators

3.1 Comparison operators and conditional execution

3.1.1 Comparison: equality operator

Question: **are two values equal**?

To ask this question, you use the == (equal equal) operator.

Don't forget this important distinction:

- **=** is an **assignment operator**, e.g., **a = b** assigns **a** with the value of **b**;
- **==** is the question *are these values equal?* so **a == b compares a** and **b**.

It is a **binary operator with left-sided binding**. It needs two arguments and **checks if they are equal**.

3.1.2 Operators

Equality: the *equal to* operator (==)

The == (equal to) operator compares the values of two operands. If they are equal, the result of the comparison is True. If they are not equal, the result of the comparison is False.

Look at the equality comparison below – what is the result of this operation?

var==0

Note that we cannot find the answer if we do not know what value is currently stored in the variable var.

If the variable has been changed many times during the execution of your program, or its initial value is entered from the console, the answer to this question can be given only by Python and only at runtime.

Now imagine a programmer who suffers from insomnia, and has to count black and white sheep separately as long as there are exactly twice as many black sheep as white ones.

The question will be as follows:

```
black_sheep == 2 * white_sheep
```

Due to the low priority of the == operator, the question shall be treated as equivalent to this one:

```
black_sheep == (2 * white_sheep)
```

So, let's practice your understanding of the == operator now – can you guess the output of the code below?

```
1  var = 0  # Assigning 0 to var
2  print(var == 0)
3
4  var = 1  # Assigning 1 to var
5  print(var == 0)
6
```

Run the code and check if you're right.

Inequality: the *not equal to* operator (!=)

The != (not equal to) operator compares the values of two operands, too. Here is the difference: if they are equal, the result of the comparison is False. If they are not equal, the result of the comparison is True.

Now take a look at the inequality comparison below – can you guess the result of this operation?

```
var = 0  # Assigning 0 to var
print(var != 0)

var = 1  # Assigning 1 to var
print(var != 0)
```

Run the code and check if you're right.

Comparison operators: greater than

You can also ask a comparison question using the > (greater than) operator.

If you want to know if there are more black sheep than white ones, you can write it as follows:

```
black_sheep > white_sheep  # Greater than
```

True confirms it; False denies it.

Comparison operators: greater than or equal to

The *greater than* operator has another special, **non-strict** variant, but it's denoted differently than in classical arithmetic notation: >= (greater than or equal to).

There are two subsequent signs, not one.

Both of these operators (strict and non-strict), as well as the two others discussed in the next section, are **binary operators with left-sided binding**, and their **priority is greater than that shown by == and !=**.

If we want to find out whether or not we have to wear a warm hat, we ask the following question:

```
centigrade_outside >= 0.0  # Greater than or equal to
```

Comparison operators: less than/less than or equal to

As you've probably already guessed, the operators used in this case are: the < (less than) operator and its non-strict sibling: <= (less than or equal to).

Look at this simple example:

```
current_velocity_mph < 85  # Less than
current_velocity_mph <= 85  # Less than or equal to
```

We're going to check if there's a risk of being fined by the highway police (the first question is strict, the second isn't).

3.1.3 Making use of the answers

What can you do with the answer (i.e., the result of a comparison operation) you get from the computer?

There are at least two possibilities: first, you can memorize it (**store it in a variable**) and make use of it later. How do you do that? Well, you use an arbitrary variable like this:

```
answer = number_of_lions >= number_of_lionesses
```

The content of the variable will tell you the answer to the question asked.

The second possibility is more convenient and far more common: you can use the answer you get to **make a decision about the future of the program**.

You need a special instruction for this purpose, and we'll discuss it very soon.

Now we need to update our **priority table**, and put all the new operators into it. It now looks as follows:

Priority	Operator	
1	+, -	Unary
2	**	
3	*, /, //, %	
4	+, -	Binary
5	<, <=, >, >=	
6	==, !=	

3.1.4 Conditions and conditional execution

You already know how to ask Python questions, but you still don't know how to make reasonable use of the answers. You have to have a mechanism which will allow you to do something **if a condition is met, and not do it if it isn't**.

It's just like in real life: you do certain things or you don't when a specific condition is met or not, e.g., you go for a walk if the weather is good, or stay home if it's wet and cold.

To make such decisions, Python offers a special instruction. Due to its nature and its application, it's called a **conditional instruction** (or conditional statement).

There are several variants of it. We'll start with the simplest, increasing the difficulty slowly.

The first form of a conditional statement, which you can see below is written very informally but figuratively:

```
if true_or_not:
    do_this_if_true
```

This conditional statement consists of the following, strictly necessary, elements in this and this order only:

- the if keyword;
- one or more white spaces;
- an expression (a question or an answer) whose value will be interpreted solely in terms of True (when its value is non-zero) and False (when it is equal to zero);
- a **colon** followed by a newline;
- an **indented** instruction or set of instructions (at least one instruction is absolutely required); the **indentation** may be achieved in two ways – by inserting a particular number of spaces (the recommendation is to use **four spaces of indentation**), or by using the *tab* character; note: if there is more than one instruction in the indented part, the indentation should be the same in all lines; even though it may look the same if you use tabs mixed with spaces, it's important to make all indentations **exactly the same** – Python 3 **does not allow the mixing of spaces and tabs** for indentation.

How does that statement work?

- If the true_or_not expression **represents the truth** (i.e., its value is not equal to zero), **the indented statement(s) will be executed**;
- if the true_or_not expression **does not represent the truth** (i.e., its value is equal to zero), **the indented statement(s) will be omitted** (ignored), and the next executed instruction will be the one after the original indentation level.

In real life, we often express a desire:

if the weather is good, we'll go for a walk

then, we'll have lunch

As you can see, having lunch is **not a conditional activity** and doesn't depend on the weather.

Knowing what conditions influence our behavior, and assuming that we have the parameterless functions go_for_a_walk() and have_lunch(), we can write the following snippet:

```
if the_weather_is_good:
    go_for_a_walk()
have_lunch()
```

Conditional execution: the if statement

If a certain sleepless Python developer falls asleep when he or she counts 120 sheep, and the sleep-inducing procedure may be implemented as a special function named sleep_and_dream(), the whole code takes the following shape:

```
if sheep_counter >= 120: # Evaluate a test expression
    sleep_and_dream() # Execute if test expression is True
```

You can read it as: if sheep_counter is greater than or equal to 120, then fall asleep and dream (i.e., execute the sleep_and_dream function.)

We've said that **conditionally executed statements have to be indented**. This creates a very legible structure, clearly demonstrating all possible execution paths in the code.

Take a look at the following code:

```
if sheep_counter >= 120:
    make_a_bed()
    take_a_shower()
    sleep_and_dream()
feed_the_sheepdogs()
```

As you can see, making a bed, taking a shower and falling asleep and dreaming are all **executed conditionally** – when sheep_counter reaches the desired limit.

Feeding the sheepdogs, however, is **always done** (i.e., the feed_the_sheepdogs() function is not indented and does not belong to the if block, which means it is always executed.)

Now we're going to discuss another variant of the conditional statement, which also allows you to perform an additional action when the condition is not met.

Conditional execution: the if-else statement

We started out with a simple phrase which read: *If the weather is good, we will go for a walk*.

Note: there is not a word about what will happen if the weather is bad. We only know that we won't go outdoors, but what we could do instead is not known. We may want to plan something in case of bad weather, too.

We can say, for example: *If the weather is good, we will go for a walk, otherwise we will go to a theater*.

Now we know what we'll do **if the conditions are met**, and we know what we'll do **if not everything goes our way**. In other words, we have a "Plan B".

Python allows us to express such alternative plans. This is done with a second, slightly more complex form of the conditional statement, the *if-else* statement:

```
if true_or_false_condition:
    perform_if_condition_true
else:
    perform_if_condition_false
```

Thus, there is a new word: else – this is a **keyword**.

The part of the code which begins with else says what to do if the condition specified for the if is not met (note the **colon** after the word).

The *if-else* execution goes as follows:

- if the condition evaluates to **True** (its value is not equal to zero), the perform_if_condition_true statement is executed, and the conditional statement comes to an end;
- if the condition evaluates to **False** (it is equal to zero), the perform_if_condition_false statement is executed, and the conditional statement comes to an end.

The if-else statement: more conditional execution

By using this form of conditional statement, we can describe our plans as follows:

```
if the_weather_is_good:
    go_for_a_walk()
else:
    go_to_a_theater()
have_lunch()
```

If the weather is good, we'll go for a walk. Otherwise, we'll go to a theater. No matter if the weather is good or bad, we'll have lunch afterwards (after the walk or after going to the theater).

Everything we've said about indentation works in the same manner inside **the *else* branch**:

```
if the_weather_is_good:
    go_for_a_walk()
    have_fun()
else:
    go_to_a_theater()
    enjoy_the_movie()
have_lunch()
```

Nested if-else statements

Now let's discuss two special cases of the conditional statement.

First, consider the case where the **instruction placed after the if is another if**.

Read what we have planned for this Sunday. If the weather is fine, we'll go for a walk. If we find a nice restaurant, we'll have lunch there. Otherwise, we'll eat a sandwich. If the weather is poor, we'll go to the theater. If there are no tickets, we'll go shopping in the nearest mall.

Let's write the same in Python. Consider carefully the code here:

```
if the_weather_is_good:
    if nice_restaurant_is_found:
        have_lunch()
    else:
        eat_a_sandwich()
else:
    if tickets_are_available:
        go_to_the_theater()
    else:
        go_shopping()
```

Here are two important points:

- this use of the if statement is known as **nesting**; remember that every else refers to the if which lies **at the same indentation level**; you need to know this to determine how the *if*s and *else*s pair up;
- consider how the **indentation improves readability**, and makes the code easier to understand and trace.

The elif statement

The second special case introduces another new Python keyword: **elif**. As you probably suspect, it's a shorter form of **else if**.

elif is used to **check more than just one condition**, and to **stop** when the first statement which is true is found.

Our next example resembles nesting, but the similarities are very slight. Again, we'll change our plans and express them as follows: If the weather is fine, we'll go for a walk, otherwise if we get tickets, we'll go to the theater, otherwise if there are free tables at the restaurant, we'll go for lunch; if all else fails, we'll stay home and play chess.

Have you noticed how many times we've used the word *otherwise*? This is the stage where the elif keyword plays its role.

Let's write the same scenario using Python:

```
if the_weather_is_good:
    go_for_a_walk()
elif tickets_are_available:
    go_to_the_theater()
elif table_is_available:
    go_for_lunch()
else:
    play_chess_at_home()
```

The way to assemble subsequent *if-elif-else* statements is sometimes called a **cascade**.

Notice again how the indentation improves the readability of the code.

Some additional attention has to be paid in this case:

- you **mustn't use else without a preceding if**;
- else is always the **last branch of the cascade**, regardless of whether you've used elif or not;
- else is an **optional** part of the cascade, and may be omitted;
- if there is an else branch in the cascade, only one of all the branches is executed;
- if there is no else branch, it's possible that none of the available branches is executed.

This may sound a little puzzling, but hopefully some simple examples will help shed more light.