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

3.2 Loops in Python 2
3.2.1 Looping your code with while 2
3.2.2 An infinite loop 3
3.2.3 The while loop: more examples 4
3.2.4 Looping your code with for 5
3.2.5 The break and continue statements 7

3.2 Loops in Python

3.2.1 Looping your code with while

Do you agree with the statement presented below?

```
while there is something to do
    do it
```

Note that this record also declares that if there is nothing to do, nothing at all will happen.

In general, in Python, a loop can be represented as follows:

```
while
    instruction
```

If you notice some similarities to the if instruction, that's quite all right. Indeed, the syntactic difference is only one: you use the word while instead of the word if.

The semantic difference is more important: when the condition is met, if performs its statements only once; while repeats the execution as long as the condition evaluates to True.

Note: all the rules regarding indentation are applicable here, too. We'll show you this soon.

Look at the algorithm below:

```
while conditional_expression:
    instruction_one
    instruction_two
    instruction_three
    :
    :
    instruction_n
```

It is now important to remember that:

if you want to execute more than one statement inside one while loop, you must (as with if) indent all the instructions in the same way;

an instruction or set of instructions executed inside the while loop is called the loop's body;

if the condition is False (equal to zero) as early as when it is tested for the first time, the body is not executed even once (note the analogy of not having to do anything if there is nothing to do);

the body should be able to change the condition's value, because if the condition is True at the beginning, the body might run continuously to infinity – notice that doing a thing usually decreases the number of things to do).

3.2.2 An infinite loop

An infinite loop, also called an **endless loop**, is a sequence of instructions in a program which repeat indefinitely (loop endlessly.)

Here's an example of a loop that is not able to finish its execution:

```
while True:
    print("I'm stuck inside a loop.")
```

This loop will infinitely print "I'm stuck inside a loop." on the screen.

Note

If you want to get the best learning experience from seeing how an infinite loop behaves, launch IDLE, create a New File, copy-paste the above code, save your file, and run the program. What you will see is the never-ending sequence of "I'm stuck inside a loop." strings printed to the Python console window. To terminate your program, just press *Ctrl-C* (or *Ctrl-Break* on some computers). This will cause a KeyboardInterrupt exception and let your program get out of the loop. We'll talk about it later in the course.

Let's go back to the sketch of the algorithm we showed you recently. We're going to show you how to use this newly learned loop to find the largest number from a large set of entered data.

Analyze the program carefully. See where the loop starts (line 8). Locate the loop's body and find out **how the body is exited**:

```
# Store the current largest number here.
largest_number = -999999999

# Input the first value.
number = int(input("Enter a number or type -1 to stop: "))

# If the number is not equal to -1, continue.
while number != -1:
    # Is number larger than largest_number?
    if number > largest_number:
        # Yes, update largest_number.
        largest_number = number
    # Input the next number.

    number = int(input("Enter a number or type -1 to stop: "))


# Print the largest number.

print("The largest number is:", largest_number)
```

Check how this code implements the algorithm we showed you earlier.

3.2.3 The while loop: more examples

Let's look at another example employing the while loop. Follow the comments to find out the idea and the solution.

```
# A program that reads a sequence of numbers

# and counts how many numbers are even and how many are odd.

# The program terminates when zero is entered.


odd_numbers = 0

even_numbers = 0


# Read the first number.

number = int(input("Enter a number or type 0 to stop: "))


# 0 terminates execution.

while number != 0:

    # Check if the number is odd.

    if number % 2 == 1:

        # Increase the odd_numbers counter.

        odd_numbers += 1

    else:

        # Increase the even_numbers counter.

        even_numbers += 1

    # Read the next number.

    number = int(input("Enter a number or type 0 to stop: "))
```

```
# Print results.
print("Odd numbers count:", odd_numbers)
print("Even numbers count:", even_numbers)
```

Certain expressions can be simplified without changing the program's behavior.

Try to recall how Python interprets the truth of a condition, and note that these two forms are equivalent:

while number != 0: and while number:.

The condition that checks if a number is odd can be coded in these equivalent forms, too:

if number % 2 == 1: and if number % 2:.

Using a counter variable to exit a loop

Look at the snippet below:

```
counter = 5
while counter != 0:
    print("Inside the loop.", counter)
    counter -= 1
print("Outside the loop.", counter)
```

This code is intended to print the string "Inside the loop." and the value stored in the counter variable during a given loop exactly five times. Once the condition has not been met (the counter variable has reached 0), the loop is exited, and the message "Outside the loop." as well as the value stored in counter is printed.

But there's one thing that can be written more compactly – the condition of the while loop.

Can you see the difference?

```
counter = 5
while counter:
    print("Inside the loop.", counter)
    counter -= 1
print("Outside the loop.", counter)
```

Is it more compact than previously? A bit. Is it more legible? That's disputable.

3.2.4 Looping your code with for

Another kind of loop available in Python comes from the observation that sometimes it's more important to **count the "turns" of the loop** than to check the conditions.

Imagine that a loop's body needs to be executed exactly one hundred times. If you would like to use the while loop to do it, it may look like this:

```
i = 0

while i < 100:

    # do_something()

    i += 1
```

It would be nice if somebody could do this boring counting for you. Is that possible?

Of course it is – there's a special loop for these kinds of tasks, and it is named for.

Actually, the for loop is designed to do more complicated tasks – **it can "browse" large collections of data item by item**. We'll show you how to do that soon, but right now we're going to present a simpler variant of its application.

Take a look at the snippet:

```
for i in range(100):

    # do_something()

    pass
```

There are some new elements. Let us tell you about them:

- the *for* keyword opens the for loop; note – there's no condition after it; you don't have to think about conditions, as they're checked internally, without any intervention;
- any variable after the *for* keyword is the **control variable** of the loop; it counts the loop's turns, and does it automatically;
- the *in* keyword introduces a syntax element describing the range of possible values being assigned to the control variable;
- the range() function (this is a very special function) is responsible for generating all the desired values of the control variable; in our example, the function will create (we can even say that it will **feed** the loop with) subsequent values from the following set: 0, 1, 2 .. 97, 98, 99; note: in this case, the range() function starts its job from 0 and finishes it one step (one integer number) before the value of its argument;
- note the *pass* keyword inside the loop body – it does nothing at all; it's an **empty instruction** – we put it here because the for loop's syntax demands at least one instruction inside the body (by the way – if, elif, else and while express the same thing)

Our next examples will be a bit more modest in the number of loop repetitions.

Take a look at the snippet below. Can you predict its output?

```
for i in range(10):

    print("The value of i is currently", i)
```

Console terminal

Run the code to check if you were right.

Note:

- the loop has been executed ten times (it's the range() function's argument)
- the last control variable's value is 9 (not 10, as **it starts from 0**, not from 1)

The range() function invocation may be equipped with two arguments, not just one:

```
for i in range(2, 8):
    print("The value of i is currently", i)
```

In this case, the first argument determines the initial (first) value of the control variable.

The last argument shows the first value the control variable will not be assigned.

Note: the range() function **accepts only integers as its arguments**, and generates sequences of integers.

Can you guess the output of the program? Run it to check if you were right now, too.

The first value shown is 2 (taken from the range()'s first argument.)

The last is 7 (although the range()'s second argument is 8).

3.2.5 The break and continue statements

So far, we've treated the body of the loop as an indivisible and inseparable sequence of instructions that are performed completely at every turn of the loop. However, as a developer, you could be faced with the following choices:

it appears that it's unnecessary to continue the loop as a whole; you should refrain from further execution of the loop's body and go further;

it appears that you need to start the next turn of the loop without completing the execution of the current turn.

Python provides two special instructions for the implementation of both these tasks. Let's say for the sake of accuracy that their existence in the language is not necessary – an experienced programmer is able to code any algorithm without these instructions. Such additions, which don't improve the language's expressive power, but only simplify the developer's work, are sometimes called syntactic candy, or syntactic sugar.

These two instructions are:

break – exits the loop immediately, and unconditionally ends the loop's operation; the program begins to execute the nearest instruction after the loop's body;

continue – behaves as if the program has suddenly reached the end of the body; the next turn is started and the condition expression is tested immediately.

Both these words are keywords.

Now we'll show you two simple examples to illustrate how the two instructions work. Look at the code in the editor. Run the program and analyze the output. Modify the code and experiment.

```
# break - example

print("The break instruction:")
for i in range(1, 6):
    if i == 3:
        break
    print("Inside the loop.", i)
print("Outside the loop.")



# continue - example

print("\nThe continue instruction:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Inside the loop.", i)
print("Outside the loop.")
```

The break and continue statements: more examples

Let's return to our program that recognizes the largest among the entered numbers. We'll convert it twice, using the break and continue instructions.

Analyze the code, and judge whether and how you would use either of them.

The break variant goes here:

```
largest_number = -99999999
counter = 0
```

```
while True:
    number = int(input("Enter a number or type -1 to end the program: "))
    if number == -1:
        break
    counter += 1
    if number > largest_number:
        largest_number = number

if counter != 0:
    print("The largest number is", largest_number)
else:
    print("You haven't entered any number.")
```

Run it, test it, and experiment with it.

And now the continue variant:

```
largest_number = -99999999
counter = 0

number = int(input("Enter a number or type -1 to end program: "))

while number != -1:
    if number == -1:
        continue
    counter += 1

    if number > largest_number:
        largest_number = number
    number = int(input("Enter a number or type -1 to end the program: "))
```

```
if counter:
    print("The largest number is", largest_number)
else:
    print("You haven't entered any number.")
```

Look carefully, the user enters the first number before the program enters the while loop. The subsequent number is entered when the program is already in the loop.

Again – run the program, test it, and experiment with it.