**Python Essentials 1:  
Module 3**

**Boolean Values, Conditional Execution, Loops, Lists and List Processing, Logical and Bitwise Operations**

In this module, you will cover the following topics:

* the Boolean data type;
* relational operators;
* making decisions in Python (if, if-else, if-elif,else)
* how to repeat code execution using loops (while, for)
* how to perform logic and bitwise operations in Python;
* lists in Python (constructing, indexing, and slicing; content manipulation)
* how to sort a list using bubble-sort algorithms;
* multidimensional lists and their applications.

# Questions and answers

A programmer writes a program and **the program asks questions**.

A computer executes the program and **provides the answers**. The program must be able to **react according to the received answers**.

Fortunately, computers know only two kinds of answers:

* yes, this is true;
* no, this is false.

You will never get a response like *Let me think....*, *I don't know*, or *Probably yes, but I don't know for sure*.

**To ask questions, Python uses a set of very special operators**. Let's go through them one after another, illustrating their effects on some simple examples.

## 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?*; 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**.

## Exercises

Now let's ask a few questions. Try to guess the answers.

**Question #1**: What is the result of the following comparison?

2 == 2    Check

True - of course, 2 is equal to 2. Python will answer True (remember this pair of predefined literals, True and False - they're Python keywords, too).

**Question #2**: What is the result of the following comparison?

2 == 2.    Check

This question is not as easy as the first one. Luckily, Python is able to convert the integer value into its real equivalent, and consequently, the answer is True.

**Question #3**: What is the result of the following comparison?

1 == 2    Check

This should be easy. The answer will be (or rather, always is) False.

# 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?

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 were 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 were 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 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).

## 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 would 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 | ==, != |  |

**LAB**

## Estimated time

5-10 minutes

## Level of difficulty

Very Easy

## Objectives

* becoming familiar with the input() function;
* becoming familiar with comparison operators in Python.

## Scenario

Using one of the comparison operators in Python, write a simple two-line program that takes the parameter n as input, which is an integer, and prints False if n is less than 100, and True if n is greater than or equal to 100.

Don't create any if blocks (we're going to talk about them very soon). Test your code using the data we've provided for you.

## Test Data

Sample input: 55

Expected output: False

Sample input: 99

Expected output: False

Sample input: 100

Expected output: True

Sample input: 101

Expected output: True

Sample input: -5

Expected output: False

Sample input: +123

Expected output: True

**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 mixing 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 theatre. No matter if the weather is good or bad, we'll have lunch afterwards (after the walk or after going to the theatre).

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 return 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.

# Analyzing code samples

Now we're going to show you some simple yet complete programs. We won't explain them in detail, because we consider the comments (and the variable names) inside the code to be sufficient guides.

All the programs solve the same problem - they **find the largest of several numbers and print it out**.

**Example 1:**

We'll start with the simplest case - **how to identify the larger of two numbers**:

# Read two numbers

number1 = int(input("Enter the first number: "))

number2 = int(input("Enter the second number: "))

# Choose the larger number

if number1 > number2:

larger\_number = number1

else:

larger\_number = number2

# Print the result

print("The larger number is:", larger\_number)

The above snippet should be clear - it reads two integer values, compares them, and finds which is the larger.

**Example 2:**

Now we're going to show you one intriguing fact. Python has an interesting feature, look at the code below:

# Read two numbers

number1 = int(input("Enter the first number: "))

number2 = int(input("Enter the second number: "))

# Choose the larger number

if number1 > number2: larger\_number = number1

else: larger\_number = number2

# Print the result

print("The larger number is:", larger\_number)

Note: if any of the *if-elif-else* branches contains just one instruction, you may code it in a more comprehensive form (you don't need to make an indented line after the keyword, but just continue the line after the colon).

This style, however, may be misleading, and we're not going to use it in our future programs, but it's definitely worth knowing if you want to read and understand someone else's programs.

There are no other differences in the code.

**Example 3:**

It's time to complicate the code - let's find the largest of three numbers. Will it enlarge the code? A bit.

We assume that the first value is the largest. Then we verify this hypothesis with the two remaining values.

Look at the code below:

# Read three numbers

number1 = int(input("Enter the first number: "))

number2 = int(input("Enter the second number: "))

number3 = int(input("Enter the third number: "))

# We temporarily assume that the first number

# is the largest one.

# We will verify this soon.

largest\_number = number1

# We check if the second number is larger than current largest\_number

# and update largest\_number if needed.

if number2 > largest\_number:

largest\_number = number2

# We check if the third number is larger than current largest\_number

# and update largest\_number if needed.

if number3 > largest\_number:

largest\_number = number3

# Print the result

print("The largest number is:", largest\_number)

This method is significantly simpler than trying to find the largest number all at once, by comparing all possible pairs of numbers (i.e., first with second, second with third, third with first). Try to rebuild the code for yourself.

# Pseudocode and introduction to loops

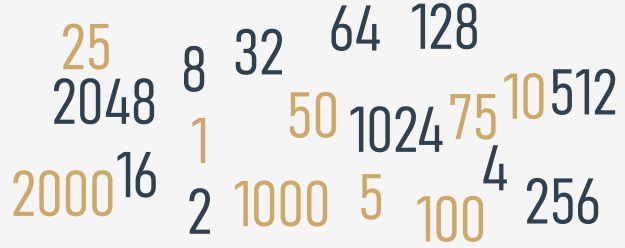
You should now be able to write a program which finds the largest of four, five, six, or even ten numbers.

You already know the scheme, so extending the size of the problem will not be particularly complex.

But what happens if we ask you to write a program that finds the largest of two hundred numbers? Can you imagine the code?

You'll need two hundred variables. If two hundred variables isn't bad enough, try to imagine searching for the largest of a million numbers.

Imagine a code that contains 199 conditional statements and two hundred invocations of the input() function. Luckily, you don't need to deal with that. There's a simpler approach.



We'll ignore the requirements of Python syntax for now, and try to analyze the problem without thinking about the real programming. In other words, we'll try to write the **algorithm**, and when we're happy with it, we'll implement it.

In this case, we'll use a kind of notation which is not an actual programming language (it can be neither compiled nor executed), but it is formalized, concise and readable. It's called **pseudocode**.

Let's look at our pseudocode below:

largest\_number = -999999999

number = int(input())

if number == -1:

print(largest\_number)

exit()

if number > largest\_number:

largest\_number = number

# Go to line 02

What's happening in it?

Firstly, we can simplify the program if, at the very beginning of the code, we assign the variable largest\_number with a value which will be smaller than any of the entered numbers. We'll use -999999999 for that purpose.

Secondly, we assume that our algorithm will not know in advance how many numbers will be delivered to the program. We expect that the user will enter as many numbers as she/he wants - the algorithm will work well with one hundred and with one thousand numbers. How do we do that?

We make a deal with the user: when the value -1 is entered, it will be a sign that there are no more data and the program should end its work.

Otherwise, if the entered value is not equal to -1, the program will read another number, and so on.

The trick is based on the assumption that any part of the code can be performed more than once - precisely, as many times as needed.

Performing a certain part of the code more than once is called a **loop**. The meaning of this term is probably obvious to you.

Lines 02 through 08 make a loop. We'll **pass through them as many times as needed** to review all the entered values.

Can you use a similar structure in a program written in Python? Yes, you can.

**Extra Info**

Python often comes with a lot of built-in functions that will do the work for you. For example, to find the largest number of all, you can use a Python built-in function called max(). You can use it with multiple arguments. Analyze the code below:

# Read three numbers.

number1 = int(input("Enter the first number: "))

number2 = int(input("Enter the second number: "))

number3 = int(input("Enter the third number: "))

# Check which one of the numbers is the greatest

# and pass it to the largest\_number variable.

largest\_number = max(number1, number2, number3)

# Print the result.

print("The largest number is:", largest\_number)

By the same fashion, you can use the min() function to return the lowest number. You can rebuild the above code and experiment with it in the Sandbox.

We're going to talk about these (and many other) functions soon. For the time being, our focus will be put on conditional execution and loops to let you gain more confidence in programming and teach you the skills that will let you fully understand and apply the two concepts in your code. So, for now, we're not taking any shortcuts.

**LAB**

## Estimated time

5-15 minutes

## Level of difficulty

Easy

## Objectives

* becoming familiar with the input() function;
* becoming familiar with comparison operators in Python;
* becoming familiar with the concept of conditional execution.

## Scenario

[Spathiphyllum](https://upload.wikimedia.org/wikipedia/commons/b/bd/Spathiphyllum_cochlearispathum_RTBG.jpg), more commonly known as a peace lily or white sail plant, is one of the most popular indoor houseplants that filters out harmful toxins from the air. Some of the toxins that it neutralizes include benzene, formaldehyde, and ammonia.

Imagine that your computer program loves these plants. Whenever it receives an input in the form of the word Spathiphyllum, it involuntarily shouts to the console the following string: "Spathiphyllum is the best plant ever!"

Write a program that utilizes the concept of conditional execution, takes a string as input, and:

* prints the sentence "Yes - Spathiphyllum is the best plant ever!" to the screen if the inputted string is "Spathiphyllum" (upper-case)
* prints "No, I want a big Spathiphyllum!" if the inputted string is "spathiphyllum" (lower-case)
* prints "Spathiphyllum! Not [input]!" otherwise. Note: [input] is the string taken as input.

Test your code using the data we've provided for you. And get yourself a Spathiphyllum, too!

## Test Data

Sample input: spathiphyllum

Expected output: No, I want a big Spathiphyllum!

Sample input: pelargonium

Expected output: Spathiphyllum! Not pelargonium!

Sample input: Spathiphyllum

Expected output: Yes - Spathiphyllum is the best plant ever!

**LAB**

## Estimated time

10-20 minutes

## Level of difficulty

Easy/Medium

## Objectives

Familiarize the student with:

* using the *if-else* instruction to branch the control path;
* building a complete program that solves simple real-life problems.

## Scenario

Once upon a time there was a land - a land of milk and honey, inhabited by happy and prosperous people. The people paid taxes, of course - their happiness had limits. The most important tax, called the *Personal Income Tax* (*PIT* for short) had to be paid once a year, and was evaluated using the following rule:

* if the citizen's income was not higher than 85,528 thalers, the tax was equal to 18% of the income minus 556 thalers and 2 cents (this was the so-called *tax relief*)
* if the income was higher than this amount, the tax was equal to 14,839 thalers and 2 cents, plus 32% of the surplus over 85,528 thalers.

Your task is to write a **tax calculator**.

* It should accept one floating-point value: the income.
* Next, it should print the calculated tax, rounded to full thalers. There's a function named round() which will do the rounding for you - you'll find it in the skeleton code in the editor.

Note: this happy country never returns money to its citizens. If the calculated tax is less than zero, it only means no tax at all (the tax is equal to zero). Take this into consideration during your calculations.

Look at the code in the editor - it only reads one input value and outputs a result, so you need to complete it with some smart calculations.

Test your code using the data we've provided.

## Test Data

Sample input: 10000

Expected output: The tax is: 1244.0 thalers

Sample input: 100000

Expected output: The tax is: 19470.0 thalers

Sample input: 1000

Expected output: The tax is: 0.0 thalers

Sample input: -100

Expected output: The tax is: 0.0 thalers

income = float(input("Enter the annual income: "))

#

# Write your code here.

#

tax = round(tax, 0)

print("The tax is:", tax, "thalers")

**LAB**

## Estimated time

10-25 minutes

## Level of difficulty

Easy/Medium

## Objectives

Familiarize the student with:

* using the if-elif-else statement;
* finding the proper implementation of verbally defined rules;
* testing code using sample input and output.

## Scenario

As you surely know, due to some astronomical reasons, years may be *leap* or *common*. The former are 366 days long, while the latter are 365 days long.

Since the introduction of the Gregorian calendar (in 1582), the following rule is used to determine the kind of year:

* if the year number isn't divisible by four, it's a *common year*;
* otherwise, if the year number isn't divisible by 100, it's a *leap year*;
* otherwise, if the year number isn't divisible by 400, it's a *common year*;
* otherwise, it's a *leap year*.

Look at the code in the editor - it only reads a year number, and needs to be completed with the instructions implementing the test we've just described.

The code should output one of two possible messages, which are Leap year or Common year, depending on the value entered.

It would be good to verify if the entered year falls into the Gregorian era, and output a warning otherwise: Not within the Gregorian calendar period. Tip: use the != and % operators.

Test your code using the data we've provided.

## Test Data

Sample input: 2000

Expected output: Leap year

Sample input: 2015

Expected output: Common year

Sample input: 1999

Expected output: Common year

Sample input: 1996

Expected output: Leap year

Sample input: 1580

Expected output: Not within the Gregorian calendar period

year = int(input("Enter a year: "))

#

# Write your code here.

#

**Key takeaways**

1. The **comparison** (otherwise known as *relational*) operators are used to compare values. The table below illustrates how the comparison operators work, assuming that x = 0, y = 1, and z = 0:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | returns True if operands' values are equal, and False otherwise |  |
| != | returns True if operands' values are not equal, and False otherwise | x != y # True  x != z # False |
| > | True if the left operand's value is greater than the right operand's value, and False otherwise | x > y # False  y > z # True |
| < | True if the left operand's value is less than the right operand's value, and False otherwise | x < y # True  y < z # False |
| ≥ | True if the left operand's value is greater than or equal to the right operand's value, and False otherwise | x >= y # False  x >= z # True  y >= z # True |
| ≤ | True if the left operand's value is less than or equal to the right operand's value, and False otherwise | x <= y # True  x <= z # True  y <= z # False |

2. When you want to execute some code only if a certain condition is met, you can use a **conditional statement**:

* a single if statement, e.g.:

x = 10

if x == 10: # condition

print("x is equal to 10") # Executed if the condition is True.

* a series of if statements, e.g.:

x = 10

if x > 5: # condition one

print("x is greater than 5") # Executed if condition one is True.

if x < 10: # condition two

print("x is less than 10") # Executed if condition two is True.

if x == 10: # condition three

print("x is equal to 10") # Executed if condition three is True.

Each if statement is tested separately.

* an if-else statement, e.g.:

x = 10

if x < 10: # Condition

print("x is less than 10") # Executed if the condition is True.

else:

print("x is greater than or equal to 10") # Executed if the condition is False.

* a series of if statements followed by an else, e.g.:

x = 10

if x > 5: # True

print("x > 5")

if x > 8: # True

print("x > 8")

if x > 10: # False

print("x > 10")

else:

print("else will be executed")

Each if is tested separately. The body of else is executed if the last if is False.

* The if-elif-else statement, e.g.:

x = 10

if x == 10: # True

print("x == 10")

if x > 15: # False

print("x > 15")

elif x > 10: # False

print("x > 10")

elif x > 5: # True

print("x > 5")

else:

print("else will not be executed")

If the condition for if is False, the program checks the conditions of the subsequent elif blocks – the first elif block that is True is executed. If all the conditions are False, the else block will be executed.

* Nested conditional statements, e.g.:

x = 10

if x > 5: # True

if x == 6: # False

print("nested: x == 6")

elif x == 10: # True

print("nested: x == 10")

else:

print("nested: else")

else:

print("else")

# Key takeaways: continued

**Exercise 1**

What is the output of the following snippet?

x = 5

y = 10

z = 8

print(x > y)

print(y > z)

Check

False

True

**output**

**Exercise 2**

What is the output of the following snippet?

x, y, z = 5, 10, 8

print(x > z)

print((y - 5) == x)

Check

False

True

**output**

**Exercise 3**

What is the output of the following snippet?

x, y, z = 5, 10, 8

x, y, z = z, y, x

print(x > z)

print((y - 5) == x)

Check

True

False

**output**

**Exercise 4**

What is the output of the following snippet?

x = 10

if x == 10:

print(x == 10)

if x > 5:

print(x > 5)

if x < 10:

print(x < 10)

else:

print("else")

Check

True

True

else

**output**

**Exercise 5**

What is the output of the following snippet?

x = "1"

if x == 1:

print("one")

elif x == "1":

if int(x) > 1:

print("two")

elif int(x) < 1:

print("three")

else:

print("four")

if int(x) == 1:

print("five")

else:

print("six")

Check

four

five

**output**

**Exercise 6**

What is the output of the following snippet?

x = 1

y = 1.0

z = "1"

if x == y:

print("one")

if y == int(z):

print("two")

elif x == y:

print("three")

else:

print("four")

Check

one

two

**output**

# 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 conditional\_expression:

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, 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).

## 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 the so-called 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.

# 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.

**REMEMBER**

Don't feel obliged to code your programs in a way that is always the shortest and the most compact. Readability may be a more important factor. Keep your code ready for a new programmer.

**LAB**

## Estimated time

15 minutes

## Level of difficulty

Easy

## Objectives

Familiarize the student with:

* using the while loop;
* reflecting real-life situations in computer code.

## Scenario

A junior magician has picked a secret number. He has hidden it in a variable named secret\_number. He wants everyone who run his program to play the *Guess the secret number* game, and guess what number he has picked for them. Those who don't guess the number will be stuck in an endless loop forever! Unfortunately, he does not know how to complete the code.

Your task is to help the magician complete the code in the editor in such a way so that the code:

* will ask the user to enter an integer number;
* will use a while loop;
* will check whether the number entered by the user is the same as the number picked by the magician. If the number chosen by the user is different than the magician's secret number, the user should see the message "Ha ha! You're stuck in my loop!" and be prompted to enter a number again. If the number entered by the user matches the number picked by the magician, the number should be printed to the screen, and the magician should say the following words: "Well done, muggle! You are free now."

The magician is counting on you! Don't disappoint him.

**EXTRA INFO**

By the way, look at the print() function. The way we've used it here is called *multi-line printing*. You can use **triple quotes** to print strings on multiple lines in order to make text easier to read, or create a special text-based design. Experiment with it.

secret\_number = 777

print(

"""

+================================+

| Welcome to my game, muggle! |

| Enter an integer number |

| and guess what number I've |

| picked for you. |

| So, what is the secret number? |

+================================+

""")

**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)

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).

# More about the for loop and the range() function with three arguments

The range() function may also accept **three arguments** - take a look at the code in the editor.

The third argument is an **increment** - it's a value added to control the variable at every loop turn (as you may suspect, the **default value of the increment is 1**).

Can you tell us how many lines will appear in the console and what values they will contain?

Run the program to find out if you were right.

You should be able to see the following lines in the console window:

The value of i is currently 2

The value of i is currently 5

**output**

Do you know why? The first argument passed to the range() function tells us what the **starting** number of the sequence is (hence 2 in the output). The second argument tells the function where to **stop** the sequence (the function generates numbers up to the number indicated by the second argument, but does not include it). Finally, the third argument indicates the **step**, which actually means the difference between each number in the sequence of numbers generated by the function.

2 (starting number) → 5 (2 increment by 3 equals 5 - the number is within the range from 2 to 8) → 8 (5 increment by 3 equals 8 - the number is not within the range from 2 to 8, because the stop parameter is not included in the sequence of numbers generated by the function.)

Note: if the set generated by the range() function is empty, the loop won't execute its body at all.

Just like here - there will be no output:

for i in range(1, 1):

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

Note: the set generated by the range() has to be sorted in **ascending order**. There's no way to force the range() to create a set in a different form when the range() function accepts exactly two arguments. This means that the range()'s second argument must be greater than the first.

Thus, there will be no output here, either:

for i in range(2, 1):

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

Let's have a look at a short program whose task is to write some of the first powers of two:

power = 1

for expo in range(16):

print("2 to the power of", expo, "is", power)

power \*= 2

The expo variable is used as a control variable for the loop, and indicates the current value of the *exponent*. The exponentiation itself is replaced by multiplying by two. Since 20 is equal to 1, then 2 × 1 is equal to 21, 2 × 21 is equal to 22, and so on. What is the greatest exponent for which our program still prints the result?

Run the code and check if the output matches your expectations.

for i in range(2, 8, 3):

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

**LAB**

## Estimated time

5-15 minutes

## Level of difficulty

Very easy

## Objectives

Familiarize the student with:

* using the for loop;
* reflecting real-life situations in computer code.

## Scenario

Do you know what Mississippi is? Well, it's the name of one of the states and rivers in the United States. The Mississippi River is about 2,340 miles long, which makes it the second longest river in the United States (the longest being the Missouri River). It's so long that a single drop of water needs 90 days to travel its entire length!

The word *Mississippi* is also used for a slightly different purpose: to *count mississippily*.

If you're not familiar with the phrase, we're here to explain to you what it means: it's used to count seconds.

The idea behind it is that adding the word *Mississippi* to a number when counting seconds aloud makes them sound closer to clock-time, and therefore "one Mississippi, two Mississippi, three Mississippi" will take approximately an actual three seconds of time! It's often used by children playing hide-and-seek to make sure the seeker does an honest count.

Your task is very simple here: write a program that uses a for loop to "count mississippily" to five. Having counted to five, the program should print to the screen the final message "Ready or not, here I come!"

Use the skeleton we've provided in the editor.

**EXTRA INFO**

Note that the code in the editor contains two elements which may not be fully clear to you at this moment: the import time statement, and the sleep() method. We're going to talk about them soon.

For the time being, we'd just like you to know that we've imported the time module and used the sleep() method to suspend the execution of each subsequent print() function inside the for loop for one second, so that the message outputted to the console resembles an actual counting. Don't worry - you'll soon learn more about modules and methods.

## Expected output

1 Mississippi

2 Mississippi

3 Mississippi

4 Mississippi

5 Mississippi

Ready or not, here I come!

import time

# Write a for loop that counts to five.

# Body of the loop - print the loop iteration number and the word "Mississippi".

# Body of the loop - use: time.sleep(1)

# Write a print function with the final message.

**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 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 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 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.

**LAB**

## Estimated time

10-20 minutes

## Level of difficulty

Easy

## Objectives

Familiarize the student with:

* using the break statement in loops;
* reflecting real-life situations in computer code.

## Scenario

The break statement is used to exit/terminate a loop.

Design a program that uses a while loop and continuously asks the user to enter a word unless the user enters "chupacabra" as the secret exit word, in which case the message "You've successfully left the loop." should be printed to the screen, and the loop should terminate.

Don't print any of the words entered by the user. Use the concept of conditional execution and the break statement.

**LAB**

## Estimated time

10-20 minutes

## Level of difficulty

Easy

## Objectives

Familiarize the student with:

* using the continue statement in loops;
* reflecting real-life situations in computer code.

## Scenario

The continue statement is used to skip the current block and move ahead to the next iteration, without executing the statements inside the loop.

It can be used with both the while and for loops.

Your task here is very special: you must design a vowel eater! Write a program that uses:

* a for loop;
* the concept of conditional execution (*if-elif-else*)
* the continue statement.

Your program must:

* ask the user to enter a word;
* use user\_word = user\_word.upper() to convert the word entered by the user to upper case; we'll talk about the so-called **string methods** and the upper() method very soon - don't worry;
* use conditional execution and the continue statement to "eat" the following vowels *A*, *E*, *I*, *O*, *U* from the inputted word;
* print the uneaten letters to the screen, each one of them on a separate line.

Test your program with the data we've provided for you.

## Test data

Sample input: Gregory

Expected output:

G

R

G

R

Y

Sample input: abstemious

Expected output:

B

S

T

M

S

Sample input: IOUEA

Expected output:

# Prompt the user to enter a word

# and assign it to the user\_word variable.

for letter in user\_word:

# Complete the body of the for loop.

**LAB**

## Estimated time

5-15 minutes

## Level of difficulty

Easy

## Objectives

Familiarize the student with:

* using the continue statement in loops;
* modifying and upgrading the existing code;
* reflecting real-life situations in computer code.

## Scenario

Your task here is even more special than before: you must redesign the (ugly) vowel eater from the previous lab (3.1.2.10) and create a better, upgraded (pretty) vowel eater! Write a program that uses:

* a for loop;
* the concept of conditional execution (*if-elif-else*)
* the continue statement.

Your program must:

* ask the user to enter a word;
* use user\_word = user\_word.upper() to convert the word entered by the user to upper case; we'll talk about the so-called **string methods** and the upper() method very soon - don't worry;
* use conditional execution and the continue statement to "eat" the following vowels *A*, *E*, *I*, *O*, *U* from the inputted word;
* assign the uneaten letters to the word\_without\_vowels variable and print the variable to the screen.

Look at the code in the editor. We've created word\_without\_vowels and assigned an empty string to it. Use concatenation operation to ask Python to combine selected letters into a longer string during subsequent loop turns, and assign it to the word\_without\_vowels variable.

Test your program with the data we've provided for you.

## Test data

Sample input: Gregory

Expected output:

GRGRY

Sample input: abstemious

Expected output:

BSTMS

Sample input: IOUEA

Expected output:

word\_without\_vowels = ""

# Prompt the user to enter a word

# and assign it to the user\_word variable.

for letter in user\_word:

# Complete the body of the loop.

# Print the word assigned to word\_without\_vowels.

# The while loop and the else branch

Both loops, while and for, have one interesting (and rarely used) feature.

We'll show you how it works - try to judge for yourself if it's usable and whether you can live without it or not.

In other words, try to convince yourself if the feature is valuable and useful, or is just syntactic sugar.

Take a look at the snippet in the editor. There's something strange at the end - the else keyword.

As you may have suspected, **loops may have the**else**branch too, like**if**s**.

The loop's else branch is **always executed once, regardless of whether the loop has entered its body or not**.

Can you guess the output? Run the program to check if you were right.

Modify the snippet a bit so that the loop has no chance to execute its body even once:

i = 5

while i < 5:

print(i)

i += 1

else:

print("else:", i)

The while's condition is False at the beginning - can you see it?

Run and test the program, and check whether the else branch has been executed or not.

i = 1

while i < 5:

print(i)

i += 1

else:

print("else:", i)

# The for loop and the else branch

for loops behave a bit differently - take a look at the snippet in the editor and run it.

The output may be a bit surprising.

The i variable retains its last value.

Modify the code a bit to carry out one more experiment.

i = 111

for i in range(2, 1):

print(i)

else:

print("else:", i)

Can you guess the output?

The loop's body won't be executed here at all. Note: we've assigned the i variable before the loop.

Run the program and check its output.

When the loop's body isn't executed, the control variable retains the value it had before the loop.

Note: **if the control variable doesn't exist before the loop starts, it won't exist when the execution reaches the**else**branch**.

How do you feel about this variant of else?

Now we're going to tell you about some other kinds of variables. Our current variables can only **store one value at a time**, but there are variables that can do much more - they can **store as many values as you want**.

for i in range(5):

print(i)

else:

print("else:", i)

**LAB**

## Estimated time

20-30 minutes

## Level of difficulty

Medium

## Objectives

Familiarize the student with:

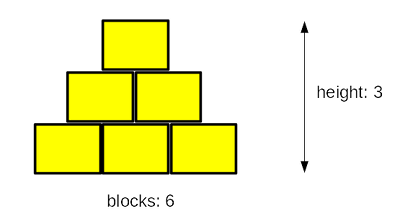
* using the while loop;
* finding the proper implementation of verbally defined rules;
* reflecting real-life situations in computer code.

## Scenario

Listen to this story: a boy and his father, a computer programmer, are playing with wooden blocks. They are building a pyramid.

Their pyramid is a bit weird, as it is actually a pyramid-shaped wall - it's flat. The pyramid is stacked according to one simple principle: each lower layer contains one block more than the layer above.

The figure illustrates the rule used by the builders:



Your task is to write a program which reads the number of blocks the builders have, and outputs the height of the pyramid that can be built using these blocks.

Note: the height is measured by the number of **fully completed layers** - if the builders don't have a sufficient number of blocks and cannot complete the next layer, they finish their work immediately.

Test your code using the data we've provided.

## Test Data

Sample input: 6

Expected output: The height of the pyramid: 3

Sample input: 20

Expected output: The height of the pyramid: 5

Sample input: 1000

Expected output: The height of the pyramid: 44

Sample input: 2

Expected output: The height of the pyramid: 1

blocks = int(input("Enter the number of blocks: "))

#

# Write your code here.

#

print("The height of the pyramid:", height)

**LAB**

## Estimated time

20 minutes

## Level of difficulty

Medium

## Objectives

Familiarize the student with:

* using the while loop;
* converting verbally defined loops into actual Python code.

## Scenario

In 1937, a German mathematician named Lothar Collatz formulated an intriguing hypothesis (it still remains unproven) which can be described in the following way:

1. take any non-negative and non-zero integer number and name it c0;
2. if it's even, evaluate a new c0 as c0 ÷ 2;
3. otherwise, if it's odd, evaluate a new c0 as 3 × c0 + 1;
4. if c0 ≠ 1, skip to point 2.

The hypothesis says that regardless of the initial value of c0, it will always go to 1.

Of course, it's an extremely complex task to use a computer in order to prove the hypothesis for any natural number (it may even require artificial intelligence), but you can use Python to check some individual numbers. Maybe you'll even find the one which would disprove the hypothesis.

Write a program which reads one natural number and executes the above steps as long as c0 remains different from 1. We also want you to count the steps needed to achieve the goal. Your code should output all the intermediate values of c0, too.

Hint: the most important part of the problem is how to transform Collatz's idea into a while loop - this is the key to success.

Test your code using the data we've provided.

## Test Data

Sample input: 15

Expected output:

46

23

70

35

106

53

160

80

40

20

10

5

16

8

4

2

1

steps = 17

Sample input: 16

Expected output:

8

4

2

1

steps = 4

Sample input: 1023

Expected output:

3070

1535

4606

2303

6910

3455

10366

5183

15550

7775

23326

11663

34990

17495

52486

26243

78730

39365

118096

59048

29524

14762

7381

22144

11072

5536

2768

1384

692

346

173

520

260

130

65

196

98

49

148

74

37

112

56

28

14

7

22

11

34

17

52

26

13

40

20

10

5

16

8

4

2

1

steps = 62

**Key takeaways**

1. There are two types of loops in Python: while and for:

* the while loop executes a statement or a set of statements as long as a specified boolean condition is true, e.g.:

# Example 1

while True:

print("Stuck in an infinite loop.")

# Example 2

counter = 5

while counter > 2:

print(counter)

counter -= 1

* the for loop executes a set of statements many times; it's used to iterate over a sequence (e.g., a list, a dictionary, a tuple, or a set - you will learn about them soon) or other objects that are iterable (e.g., strings). You can use the for loop to iterate over a sequence of numbers using the built-in range function. Look at the examples below:

# Example 1

word = "Python"

for letter in word:

print(letter, end="\*")

# Example 2

for i in range(1, 10):

if i % 2 == 0:

print(i)

2. You can use the break and continue statements to change the flow of a loop:

* You use break to exit a loop, e.g.:

text = "OpenEDG Python Institute"

for letter in text:

if letter == "P":

break

print(letter, end="")

* You use continue to skip the current iteration, and continue with the next iteration, e.g.:

text = "pyxpyxpyx"

for letter in text:

if letter == "x":

continue

print(letter, end="")

3. The while and for loops can also have an else clause in Python. The else clause executes after the loop finishes its execution as long as it has not been terminated by break, e.g.:

n = 0

while n != 3:

print(n)

n += 1

else:

print(n, "else")

print()

for i in range(0, 3):

print(i)

else:

print(i, "else")

4. The range() function generates a sequence of numbers. It accepts integers and returns range objects. The syntax of range() looks as follows: range(start, stop, step), where:

* start is an optional parameter specifying the starting number of the sequence (0 by default)
* stop is an optional parameter specifying the end of the sequence generated (it is not included),
* and step is an optional parameter specifying the difference between the numbers in the sequence (1 by default.)

Example code:

for i in range(3):

print(i, end=" ") # Outputs: 0 1 2

for i in range(6, 1, -2):

print(i, end=" ") # Outputs: 6, 4, 2

# Key takeaways: continued

**Exercise 1**

Create a for loop that counts from 0 to 10, and prints odd numbers to the screen. Use the skeleton below:

for i in range(1, 11):

# Line of code.

# Line of code.

Check

Sample solution:  
for i in range(0, 11):

if i % 2 != 0:

print(i)

**Exercise 2**

Create a while loop that counts from 0 to 10, and prints odd numbers to the screen. Use the skeleton below:

x = 1

while x < 11:

# Line of code.

# Line of code.

# Line of code.

Check

Sample solution:  
x = 1

while x < 11:

if x % 2 != 0:

print(x)

x += 1

**Exercise 3**

Create a program with a for loop and a break statement. The program should iterate over characters in an email address, exit the loop when it reaches the @ symbol, and print the part before @ on one line. Use the skeleton below:

for ch in "john.smith@pythoninstitute.org":

if ch == "@":

# Line of code.

# Line of code.

Check

Sample solution:  
for ch in "john.smith@pythoninstitute.org":

if ch == "@":

break

print(ch, end="")

**Exercise 4**

Create a program with a for loop and a continue statement. The program should iterate over a string of digits, replace each 0 with x, and print the modified string to the screen. Use the skeleton below:

for digit in "0165031806510":

if digit == "0":

# Line of code.

# Line of code.

# Line of code.

Check

Sample solution:  
for digit in "0165031806510":

if digit == "0":

print("x", end="")

continue

print(digit, end="")

**Exercise 5**

What is the output of the following code?

n = 3

while n > 0:

print(n + 1)

n -= 1

else:

print(n)

Check

4

3

2

0

**Exercise 6**

What is the output of the following code?

n = range(4)

for num in n:

print(num - 1)

else:

print(num)

Check

-1

0

1

2

3

**Exercise 7**

What is the output of the following code?

for i in range(0, 6, 3):

print(i)

Check

0

3

# Computer logic

Have you noticed that the conditions we've used so far have been very simple, not to say, quite primitive? The conditions we use in real life are much more complex. Let's look at this sentence:

*If we have some free time, and the weather is good, we will go for a walk.*

We've used the conjunction and, which means that going for a walk depends on the simultaneous fulfilment of these two conditions. In the language of logic, such a connection of conditions is called a **conjunction**. And now another example:

*If you are in the mall or I am in the mall, one of us will buy a gift for Mom.*

The appearance of the word or means that the purchase depends on at least one of these conditions. In logic, such a compound is called a **disjunction**.

It's clear that Python must have operators to build conjunctions and disjunctions. Without them, the expressive power of the language would be substantially weakened. They're called **logical operators**.

## and

One logical conjunction operator in Python is the word *and*. It's a **binary operator with a priority that is lower than the one expressed by the comparison operators**. It allows us to code complex conditions without the use of parentheses like this one:

counter > 0 and value == 100

The result provided by the and operator can be determined on the basis of the **truth table**.

If we consider the conjunction of A and B, the set of possible values of arguments and corresponding values of the conjunction looks as follows:

|  |  |  |
| --- | --- | --- |
| **Argument**A | **Argument**B | A and B |
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

## or

A disjunction operator is the word or. It's a **binary operator with a lower priority than**and (just like + compared to \*). Its truth table is as follows:

|  |  |  |
| --- | --- | --- |
| **Argument**A | **Argument**B | A or B |
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

## not

In addition, there's another operator that can be applied for constructing conditions. It's a **unary operator performing a logical negation**. Its operation is simple: it turns truth into falsehood and falsehood into truth.

This operator is written as the word not, and its **priority is very high: the same as the unary**+**and**-. Its truth table is simple:

|  |  |
| --- | --- |
| **Argument** | not**Argument** |
| False | True |
| True | False |

# Logical expressions

Let's create a variable named var and assign 1 to it. The following conditions are **pairwise equivalent**:

# Example 1:

print(var > 0)

print(not (var <= 0))

# Example 2:

print(var != 0)

print(not (var == 0))

You may be familiar with De Morgan's laws. They say that:

*The negation of a conjunction is the disjunction of the negations.*

*The negation of a disjunction is the conjunction of the negations.*

Let's write the same thing using Python:

not (p and q) == (not p) or (not q)

not (p or q) == (not p) and (not q)

Note how the parentheses have been used to code the expressions - we put them there to improve readability.

We should add that none of these two-argument operators can be used in the abbreviated form known as op=. This exception is worth remembering.

## Logical values vs. single bits

Logical operators take their arguments as a whole regardless of how many bits they contain. The operators are aware only of the value: zero (when all the bits are reset) means False; not zero (when at least one bit is set) means True.

The result of their operations is one of these values: False or True. This means that this snippet will assign the value True to the j variable if i is not zero; otherwise, it will be False.

i = 1

j = not not i

# Bitwise operators

However, there are four operators that allow you to **manipulate single bits of data**. They are called **bitwise operators**.

They cover all the operations we mentioned before in the logical context, and one additional operator. This is the xor (as in **exclusive or**) operator, and is denoted as ^ (caret).

Here are all of them:

* & (ampersand) - bitwise conjunction;
* | (bar) - bitwise disjunction;
* ~ (tilde) - bitwise negation;
* ^ (caret) - bitwise exclusive or (xor).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bitwise operations (**&**,**|**, and**^**)** | | | | |
| **Argument**A | **Argument**B | A & B | A | B | A ^ B |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

|  |  |
| --- | --- |
| **Bitwise operations (~)** | |
| **Argument** | ~**Argument** |
| 0 | 1 |
| 1 | 0 |

Let's make it easier:

* & requires exactly two 1s to provide 1 as the result;
* | requires at least one 1 to provide 1 as the result;
* ^ requires exactly one 1 to provide 1 as the result.

Let us add an important remark: the arguments of these operators **must be integers**; we must not use floats here.

The difference in the operation of the logical and bit operators is important: **the logical operators do not penetrate into the bit level of its argument**. They're only interested in the final integer value.

Bitwise operators are stricter: they deal with **every bit separately**. If we assume that the integer variable occupies 64 bits (which is common in modern computer systems), you can imagine the bitwise operation as a 64-fold evaluation of the logical operator for each pair of bits of the arguments. This analogy is obviously imperfect, as in the real world all these 64 operations are performed at the same time (simultaneously).

# Logical vs. bit operations: continued

We'll now show you an example of the difference in operation between the logical and bit operations. Let's assume that the following assignments have been performed:

i = 15

j = 22

If we assume that the integers are stored with 32 bits, the bitwise image of the two variables will be as follows:

i: 00000000000000000000000000001111

j: 00000000000000000000000000010110

The assignment is given:

log = i and j

We are dealing with a logical conjunction here. Let's trace the course of the calculations. Both variables i and j are not zeros, so will be deemed to represent True. Consulting the truth table for the and operator, we can see that the result will be True. No other operations are performed.

log: True

Now the bitwise operation - here it is:

bit = i & j

The & operator will operate with each pair of corresponding bits separately, producing the values of the relevant bits of the result. Therefore, the result will be as follows:

|  |  |
| --- | --- |
| i | 00000000000000000000000000001111 |
| j | 00000000000000000000000000010110 |
| bit = i & j | 00000000000000000000000000000110 |

These bits correspond to the integer value of six.

Let's look at the negation operators now. First the logical one:

logneg = not i

The logneg variable will be set to False - nothing more needs to be done.

The bitwise negation goes like this:

bitneg = ~i

It may be a bit surprising: the bitneg variable value is -16. This may seem strange, but isn't at all. If you wish to learn more, you should check out the binary numeral system and the rules governing two's complement numbers.

|  |  |
| --- | --- |
| i | 00000000000000000000000000001111 |
| bitneg = ~i | 11111111111111111111111111110000 |

Each of these two-argument operators can be used in **abbreviated form**. These are the examples of their equivalent notations:

|  |  |
| --- | --- |
| x = x & y | x &= y |
| x = x | y | x |= y |
| x = x ^ y | x ^= y |

**How do we deal with single bits?**

We'll now show you what you can use bitwise operators for. Imagine that you're a developer obliged to write an important piece of an operating system. You've been told that you're allowed to use a variable assigned in the following way:

flag\_register = 0x1234

The variable stores the information about various aspects of system operation. **Each bit of the variable stores one yes/no value**. You've also been told that only one of these bits is yours - the third (remember that bits are numbered from zero, and bit number zero is the lowest one, while the highest is number 31). The remaining bits are not allowed to change, because they're intended to store other data. Here's your bit marked with the letter x:

flag\_register = 0000000000000000000000000000x000

You may be faced with the following tasks:

1. **Check the state of your bit** - you want to find out the value of your bit; comparing the whole variable to zero will not do anything, because the remaining bits can have completely unpredictable values, but you can use the following conjunction property:

x & 1 = x

x & 0 = 0

If you apply the & operation to the flag\_register variable along with the following bit image:

00000000000000000000000000001000

(note the 1 at your bit's position) as the result, you obtain one of the following bit strings:

* 00000000000000000000000000001000 if your bit was set to 1
* 0000000000000000000000000000000 if your bit was reset to 0

Such a sequence of zeros and ones, whose task is to grab the value or to change the selected bits, is called a **bit mask**.

Let's build a bit mask to detect the state of your bit. It should point to **the third bit**. That bit has the weight of 23 = 8. A suitable mask could be created by the following declaration:

the\_mask = 8

You can also make a sequence of instructions depending on the state of your bit i here it is:

if flag\_register & the\_mask:

# My bit is set.

else:

# My bit is reset.

2. **Reset your bit** - you assign a zero to the bit while all the other bits must remain unchanged; let's use the same property of the conjunction as before, but let's use a slightly different mask - exactly as below:

11111111111111111111111111110111

Note that the mask was created as a result of the negation of all the bits of the\_mask variable. Resetting the bit is simple, and looks like this (choose the one you like more):

flag\_register = flag\_register & ~the\_mask

flag\_register &= ~the\_mask

3. **Set your bit** - you assign a 1 to your bit, while all the remaining bits must remain unchanged; use the following disjunction property:

x | 1 = 1

x | 0 = x

You're now ready to set your bit with one of the following instructions:

flag\_register = flag\_register | the\_mask

flag\_register |= the\_mask

4. **Negate your bit** - you replace a 1 with a 0 and a 0 with a 1. You can use an interesting property of the xor operator:

x ^ 1 = ~x

x ^ 0 = x

and negate your bit with the following instructions:

flag\_register = flag\_register ^ the\_mask

flag\_register ^= the\_mask

**Binary left shift and binary right shift**

Python offers yet another operation relating to single bits: **shifting**. This is applied only to **integer** values, and you mustn't use floats as arguments for it.

You already apply this operation very often and quite unconsciously. How do you multiply any number by ten? Take a look:

12345 × 10 = 123450

As you can see, **multiplying by ten is in fact a shift** of all the digits to the left and filling the resulting gap with zero.

Division by ten? Take a look:

12340 ÷ 10 = 1234

Dividing by ten is nothing but shifting the digits to the right.

The same kind of operation is performed by the computer, but with one difference: as two is the base for binary numbers (not 10), **shifting a value one bit to the left thus corresponds to multiplying it by two**; respectively, **shifting one bit to the right is like dividing by two** (notice that the rightmost bit is lost).

The **shift operators** in Python are a pair of **digraphs**: << and >>, clearly suggesting in which direction the shift will act.

value << bits

value >> bits

**The left argument of these operators is an integer value whose bits are shifted. The right argument determines the size of the shift.**

It shows that this operation is certainly not commutative.

The priority of these operators is very high. You'll see them in the updated table of priorities, which we'll show you at the end of this section.

Take a look at the shifts in the editor window.

The final print() invocation produces the following output:

17 68 8

**output**

Note:

* 17 >> 1 → 17 // 2 (**17** floor-divided by **2 to the power of 1**) → 8 (shifting to the right by one bit is the same as integer division by two)
* 17 << 2 → 17 \* 4 (**17** multiplied by **2 to the power of 2**) → 68 (shifting to the left by two bits is the same as integer multiplication by four)

And here is the **updated priority table**, containing all the operators introduced so far:

|  |  |  |
| --- | --- | --- |
| **Priority** | **Operator** |  |
| 1 | ~, +, - | unary |
| 2 | \*\* |  |
| 3 | \*, /, //, % |  |
| 4 | +, - | binary |
| 5 | <<, >> |  |
| 6 | <, <=, >, >= |  |
| 7 | ==, != |  |
| 8 | & |  |
| 9 | | |  |
| 10 | =, +=, -=, \*=, /=, %=, &=, ^=, |=, >>=, <<= |  |

var = 17

var\_right = var >> 1

var\_left = var << 2

print(var, var\_left, var\_right)

**Key takeaways**

1. Python supports the following logical operators:

* and → if both operands are true, the condition is true, e.g., (True and True) is True,
* or → if any of the operands are true, the condition is true, e.g., (True or False) is True,
* not → returns false if the result is true, and returns true if the result is false, e.g., not True is False.

2. You can use bitwise operators to manipulate single bits of data. The following sample data:

* x = 15, which is 0000 1111 in binary,
* y = 16, which is 0001 0000 in binary.

will be used to illustrate the meaning of bitwise operators in Python. Analyze the examples below:

* & does a *bitwise and*, e.g., x & y = 0, which is 0000 0000 in binary,
* | does a *bitwise or*, e.g., x | y = 31, which is 0001 1111 in binary,
* ˜  does a *bitwise not*, e.g., ˜ x = 240\*, which is 1111 0000 in binary,
* ^ does a *bitwise xor*, e.g., x ^ y = 31, which is 0001 1111 in binary,
* >> does a *bitwise right shift*, e.g., y >> 1 = 8, which is 0000 1000 in binary,
* << does a *bitwise left shift*, e.g., y << 3 = , which is 1000 0000 in binary,

\* -16 (decimal from signed 2's complement) -- read more about the [Two's complement](https://en.wikipedia.org/wiki/Two%27s_complement) operation.

**Exercise 1**

What is the output of the following snippet?

x = 1

y = 0

z = ((x == y) and (x == y)) or not(x == y)

print(not(z))

Check

False

**Exercise 2**

What is the output of the following snippet?

x = 4

y = 1

a = x & y

b = x | y

c = ~x # tricky!

d = x ^ 5

e = x >> 2

f = x << 2

print(a, b, c, d, e, f)

Check

0 5 -5 1 1 16

**Why do we need lists?**

It may happen that you have to read, store, process, and finally, print dozens, maybe hundreds, perhaps even thousands of numbers. What then? Do you need to create a separate variable for each value? Will you have to spend long hours writing statements like the one below?

var1 = int(input())

var2 = int(input())

var3 = int(input())

var4 = int(input())

var5 = int(input())

var6 = int(input())

:

:

If you don't think that this is a complicated task, then take a piece of paper and write a program that:

* reads five numbers,
* prints them in order from the smallest to the largest (NB, this kind of processing is called **sorting**).

You should find that you don't even have enough paper to complete the task.

So far, you've learned how to declare variables that are able to store exactly one given value at a time. Such variables are sometimes called **scalars** by analogy with mathematics. All the variables you've used so far are actually scalars.

Think of how convenient it would be to declare a variable that could **store more than one value**. For example, a hundred, or a thousand or even ten thousand. It would still be one and the same variable, but very wide and capacious. Sounds appealing? Perhaps, but how would it handle such a container full of different values? How would it choose just the one you need?

What if you could just number them? And then say: *give me the value number 2; assign the value number 15; increase the value number 10000*.

We'll show you how to declare such **multi-value variables**. We'll do this with the example we just suggested. We'll write a **program that sorts a sequence of numbers**. We won't be particularly ambitious - we'll assume that there are exactly five numbers.

Let's create a variable called numbers; it's assigned with not just one number, but is filled with a list consisting of five values (note: the **list starts with an open square bracket and ends with a closed square bracket**; the space between the brackets is filled with five numbers separated by commas).

numbers = [10, 5, 7, 2, 1]

Let's say the same thing using adequate terminology: numbers**is a list consisting of five values, all of them numbers**. We can also say that this statement creates a list of length equal to five (as in there are five elements inside it).

The elements inside a list **may have different types**. Some of them may be integers, others floats, and yet others may be lists.

Python has adopted a convention stating that the elements in a list are **always numbered starting from zero**. This means that the item stored at the beginning of the list will have the number zero. Since there are five elements in our list, the last of them is assigned the number four. Don't forget this.

You'll soon get used to it, and it'll become second nature.

Before we go any further in our discussion, we have to state the following: our **list is a collection of elements, but each element is a scalar**.

# Indexing lists

How do you change the value of a chosen element in the list?

Let's **assign a new value of**111**to the first element** in the list. We do it this way:

numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers) # Printing original list content.

numbers[0] = 111

print("New list content: ", numbers) # Current list content.

And now we want **the value of the fifth element to be copied to the second element** - can you guess how to do it?

numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers) # Printing original list content.

numbers[0] = 111

print("\nPrevious list content:", numbers) # Printing previous list content.

numbers[1] = numbers[4] # Copying value of the fifth element to the second.

print("New list content:", numbers) # Printing current list content.

The value inside the brackets which selects one element of the list is called an **index**, while the operation of selecting an element from the list is known as **indexing**.

We're going to use the print() function to print the list content each time we make the changes. This will help us follow each step more carefully and see what's going on after a particular list modification.

Note: all the indices used so far are literals. Their values are fixed at runtime, but **any expression can be the index**, too. This opens up lots of possibilities.

numbers = [10, 5, 7, 2, 1]

print("List content:", numbers) # Printing original list content.

# Accessing list content

Each of the list's elements may be accessed separately. For example, it can be printed:

print(numbers[0]) # Accessing the list's first element.

Assuming that all of the previous operations have been completed successfully, the snippet will send 111 to the console.

As you can see in the editor, the list may also be printed as a whole - just like here:

print(numbers) # Printing the whole list.

As you've probably noticed before, Python decorates the output in a way that suggests that all the presented values form a list. The output from the example snippet above looks like this:

[111, 1, 7, 2, 1]

**output**

## The len() function

The **length of a list** may vary during execution. New elements may be added to the list, while others may be removed from it. This means that the list is a very dynamic entity.

If you want to check the list's current length, you can use a function named len() (its name comes from *length*).

The function takes the **list's name as an argument, and returns the number of elements currently stored** inside the list (in other words - the list's length).

Look at the last line of code in the editor, run the program and check what value it will print to the console. Can you guess?

numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers) # Printing original list content.

numbers[0] = 111

print("\nPrevious list content:", numbers) # Printing previous list content.

numbers[1] = numbers[4] # Copying value of the fifth element to the second.

print("Previous list content:", numbers) # Printing previous list content.

print("\nList length:", len(numbers)) # Printing the list's length.

# Removing elements from a list

Any of the list's elements may be **removed** at any time - this is done with an instruction named del (delete). Note: it's an **instruction**, not a function.

You have to point to the element to be removed - it'll vanish from the list, and the list's length will be reduced by one.

Look at the snippet below. Can you guess what output it will produce? Run the program in the editor and check.

del numbers[1]

print(len(numbers))

print(numbers)

**You can't access an element which doesn't exist** - you can neither get its value nor assign it a value. Both of these instructions will cause runtime errors now:

print(numbers[4])

numbers[4] = 1

Add the snippet above after the last line of code in the editor, run the program and check what happens.

Note: we've removed one of the list's elements - there are only four elements in the list now. This means that element number four doesn't exist.

numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers) # Printing original list content.

numbers[0] = 111

print("\nPrevious list content:", numbers) # Printing previous list content.

numbers[1] = numbers[4] # Copying value of the fifth element to the second.

print("Previous list content:", numbers) # Printing previous list content.

print("\nList's length:", len(numbers)) # Printing previous list length.

###

del numbers[1] # Removing the second element from the list.

print("New list's length:", len(numbers)) # Printing new list length.

print("\nNew list content:", numbers) # Printing current list content.

###

# Negative indices are legal

It may look strange, but negative indices are legal, and can be very useful.

An element with an index equal to -1 is **the last one in the list**.

print(numbers[-1])

The example snippet will output 1. Run the program and check.

Similarly, the element with an index equal to -2 is **the one before last in the list**.

print(numbers[-2])

The example snippet will output 2.

The last accessible element in our list is numbers[-4] (the first one) - don't try to go any further!

numbers = [111, 7, 2, 1]

print(numbers[-1])

print(numbers[-2])

**LAB**

## Estimated time

5 minutes

## Level of difficulty

Very easy

## Objectives

Familiarize the student with:

* using basic instructions related to lists;
* creating and modifying lists.

## Scenario

There once was a hat. The hat contained no rabbit, but a list of five numbers: 1, 2, 3, 4, and 5.

Your task is to:

* write a line of code that prompts the user to replace the middle number in the list with an integer number entered by the user (Step 1)
* write a line of code that removes the last element from the list (Step 2)
* write a line of code that prints the length of the existing list (Step 3).

Ready for this challenge?

hat\_list = [1, 2, 3, 4, 5] # This is an existing list of numbers hidden in the hat.

# Step 1: write a line of code that prompts the user

# to replace the middle number with an integer number entered by the user.

# Step 2: write a line of code that removes the last element from the list.

# Step 3: write a line of code that prints the length of the existing list.

print(hat\_list)

# Functions vs. methods

A **method is a specific kind of function** - it behaves like a function and looks like a function, but differs in the way in which it acts, and in its invocation style.

A **function doesn't belong to any data** - it gets data, it may create new data and it (generally) produces a result.

A method does all these things, but is also able to **change the state of a selected entity**.

**A method is owned by the data it works for, while a function is owned by the whole code**.

This also means that invoking a method requires some specification of the data from which the method is invoked.

It may sound puzzling here, but we'll deal with it in depth when we delve into object-oriented programming.

In general, a typical function invocation may look like this:

result = function(arg)

The function takes an argument, does something, and returns a result.

A typical method invocation usually looks like this:

result = data.method(arg)

Note: the name of the method is preceded by the name of the data which owns the method. Next, you add a **dot**, followed by the **method name**, and a pair of **parenthesis enclosing the arguments**.

The method will behave like a function, but can do something more - it can **change the internal state of the data** from which it has been invoked.

You may ask: why are we talking about methods, not about lists?

This is an essential issue right now, as we're going to show you how to add new elements to an existing list. This can be done with methods owned by all the lists, not by functions.

**Adding elements to a list: append() and insert()**

A new element may be *glued* to the end of the existing list:

list.append(value)

Such an operation is performed by a method named append(). It takes its argument's value and puts it **at the end of the list** which owns the method.

The list's length then increases by one.

The insert() method is a bit smarter - it can add a new element **at any place in the list**, not only at the end.

list.insert(location, value)

It takes two arguments:

* the first shows the required location of the element to be inserted; note: all the existing elements that occupy locations to the right of the new element (including the one at the indicated position) are shifted to the right, in order to make space for the new element;
* the second is the element to be inserted.

Look at the code in the editor. See how we use the append() and insert() methods. Pay attention to what happens after using insert(): the former first element is now the second, the second the third, and so on.

Add the following snippet after the last line of code in the editor:

numbers.insert(1, 333)

Print the final list content to the screen and see what happens. The snippet above snippet inserts 333 into the list, making it the second element. The former second element becomes the third, the third the fourth, and so on.

numbers = [111, 7, 2, 1]

print(len(numbers))

print(numbers)

###

numbers.append(4)

print(len(numbers))

print(numbers)

###

numbers.insert(0, 222)

print(len(numbers))

print(numbers)

#

# Adding elements to a list: continued

You can **start a list's life by making it empty** (this is done with an empty pair of square brackets) and then adding new elements to it as needed.

Take a look at the snippet in the editor. Try to guess its output after the for loop execution. Run the program to check if you were right.

It'll be a sequence of consecutive integer numbers from 1 (you then add one to all the appended values) to 5.

We've modified the snippet a bit:

my\_list = [] # Creating an empty list.

for i in range(5):

my\_list.insert(0, i + 1)

print(my\_list)

what will happen now? Run the program and check if this time you were right, too.

You should get the same sequence, but in **reverse order** (this is the merit of using the insert() method).

my\_list = [] # Creating an empty list.

for i in range(5):

my\_list.append(i + 1)

print(my\_list)

# Making use of lists

The for loop has a very special variant that can **process lists** very effectively - let's take a look at that.

Let's assume that you want to **calculate the sum of all the values stored in the**my\_list**list**.

You need a variable whose sum will be stored and initially assigned a value of 0 - its name will be total. (Note: we're not going to name it sum as Python uses the same name for one of its built-in functions - sum(). **Using the same name would generally be considered a bad practice**.) Then you add to it all the elements of the list using the for loop. Take a look at the snippet in the editor.

Let's comment on this example:

* the list is assigned a sequence of five integer values;
* the i variable takes the values 0, 1, 2, 3, and 4, and then it indexes the list, selecting the subsequent elements: the first, second, third, fourth and fifth;
* each of these elements is added together by the += operator to the total variable, giving the final result at the end of the loop;
* note the way in which the len() function has been employed - it makes the code independent of any possible changes in the list's content.

## The second face of the for loop

But the for loop can do much more. It can hide all the actions connected to the list's indexing, and deliver all the list's elements in a handy way.

This modified snippet shows how it works:

my\_list = [10, 1, 8, 3, 5]

total = 0

for i in my\_list:

total += i

print(total)

What happens here?

* the for instruction specifies the variable used to browse the list (i here) followed by the in keyword and the name of the list being processed (my\_list here)
* the i variable is assigned the values of all the subsequent list's elements, and the process occurs as many times as there are elements in the list;
* this means that you use the i variable as a copy of the elements' values, and you don't need to use indices;
* the len() function is not needed here, either.

my\_list = [10, 1, 8, 3, 5]

total = 0

for i in range(len(my\_list)):

total += my\_list[i]

print(total)

# Lists in action

Let's leave lists aside for a short moment and look at one intriguing issue.

Imagine that you need to rearrange the elements of a list, i.e., reverse the order of the elements: the first and the fifth as well as the second and fourth elements will be swapped. The third one will remain untouched.

Question: how can you swap the values of two variables?

Take a look at the snippet:

variable\_1 = 1

variable\_2 = 2

variable\_2 = variable\_1

variable\_1 = variable\_2

If you do something like this, you would **lose the value previously stored** in variable\_2. Changing the order of the assignments will not help. You need a **third variable that serves as an auxiliary storage**.

This is how you can do it:

variable\_1 = 1

variable\_2 = 2

auxiliary = variable\_1

variable\_1 = variable\_2

variable\_2 = auxiliary

Python offers a more convenient way of doing the swap - take a look:

variable\_1 = 1

variable\_2 = 2

variable\_1, variable\_2 = variable\_2, variable\_1

Clear, effective and elegant - isn't it?

**Lists in action**

Now you can easily **swap** the list's elements to **reverse their order**:

my\_list = [10, 1, 8, 3, 5]

my\_list[0], my\_list[4] = my\_list[4], my\_list[0]

my\_list[1], my\_list[3] = my\_list[3], my\_list[1]

print(my\_list)

Run the snippet. Its output should look like this:

[5, 3, 8, 1, 10]

**output**

It looks fine with five elements.

Will it still be acceptable with a list containing 100 elements? No, it won't.

Can you use the for loop to do the same thing automatically, irrespective of the list's length? Yes, you can.

This is how we've done it:

my\_list = [10, 1, 8, 3, 5]

length = len(my\_list)

for i in range(length // 2):

my\_list[i], my\_list[length - i - 1] = my\_list[length - i - 1], my\_list[i]

print(my\_list)

Note:

* we've assigned the length variable with the current list's length (this makes our code a bit clearer and shorter)
* we've launched the for loop to run through its body length // 2 times (this works well for lists with both even and odd lengths, because when the list contains an odd number of elements, the middle one remains untouched)
* we've swapped the ith element (from the beginning of the list) with the one with an index equal to (length - i - 1) (from the end of the list); in our example, for i equal to 0 the (length - i - 1) gives 4; for i equal to 1, it gives 3 ‒ this is exactly what we needed.

Lists are extremely useful, and you'll encounter them very often.

**LAB**

## Estimated time

10-15 minutes

## Level of difficulty

Easy

## Objectives

Familiarize the student with:

* creating and modifying simple lists;
* using methods to modify lists.

## Scenario

The Beatles were one of the most popular music group of the 1960s, and the best-selling band in history. Some people consider them to be the most influential act of the rock era. Indeed, they were included in *Time* magazine's compilation of the 20th Century's 100 most influential people.

The band underwent many line-up changes, culminating in 1962 with the line-up of John Lennon, Paul McCartney, George Harrison, and Richard Starkey (better known as Ringo Starr).

Write a program that reflects these changes and lets you practice with the concept of lists. Your task is to:

* step 1: create an empty list named beatles;
* step 2: use the append() method to add the following members of the band to the list: John Lennon, Paul McCartney, and George Harrison;
* step 3: use the for loop and the append() method to prompt the user to add the following members of the band to the list: Stu Sutcliffe, and Pete Best;
* step 4: use the del instruction to remove Stu Sutcliffe and Pete Best from the list;
* step 5: use the insert() method to add Ringo Starr to the beginning of the list.

By the way, are you a Beatles fan? (The Beatles is one of Greg's favorite bands. But wait...who's Greg...?)

# step 1

print("Step 1:", beatles)

# step 2

print("Step 2:", beatles)

# step 3

print("Step 3:", beatles)

# step 4

print("Step 4:", beatles)

# step 5

print("Step 5:", beatles)

# testing list legth

print("The Fab", len(beatles))

# Key takeaways

1. The **list is a type of data** in Python used to **store multiple objects**. It is an **ordered and mutable collection** of comma-separated items between square brackets, e.g.:

my\_list = [1, None, True, "I am a string", 256, 0]

2. Lists can be **indexed and updated**, e.g.:

my\_list = [1, None, True, 'I am a string', 256, 0]

print(my\_list[3]) # outputs: I am a string

print(my\_list[-1]) # outputs: 0

my\_list[1] = '?'

print(my\_list) # outputs: [1, '?', True, 'I am a string', 256, 0]

my\_list.insert(0, "first")

my\_list.append("last")

print(my\_list) # outputs: ['first', 1, '?', True, 'I am a string', 256, 0, 'last']

3. Lists can be **nested**, e.g.:

my\_list = [1, 'a', ["list", 64, [0, 1], False]]

You will learn more about nesting in module 3.1.7 - for the time being, we just want you to be aware that something like this is possible, too.

4. List elements and lists can be **deleted**, e.g.:

my\_list = [1, 2, 3, 4]

del my\_list[2]

print(my\_list) # outputs: [1, 2, 4]

del my\_list # deletes the whole list

Again, you will learn more about this in module 3.1.6 - don't worry. For the time being just try to experiment with the above code and check how changing it affects the output.

5. Lists can be **iterated** through using the for loop, e.g.:

my\_list = ["white", "purple", "blue", "yellow", "green"]

for color in my\_list:

print(color)

6. The len() function may be used to **check the list's length**, e.g.:

my\_list = ["white", "purple", "blue", "yellow", "green"]

print(len(my\_list)) # outputs 5

del my\_list[2]

print(len(my\_list)) # outputs 4

7. A typical **function** invocation looks as follows: result = function(arg), while a typical **method** invocation looks like this:result = data.method(arg).

**Exercise 1**

What is the output of the following snippet?

lst = [1, 2, 3, 4, 5]

lst.insert(1, 6)

del lst[0]

lst.append(1)

print(lst)

Check

[6, 2, 3, 4, 5, 1]

**Exercise 2**

What is the output of the following snippet?

lst = [1, 2, 3, 4, 5]

lst\_2 = []

add = 0

for number in lst:

add += number

lst\_2.append(add)

print(lst\_2)

Check

[1, 3, 6, 10, 15]

**Exercise 3**

What happens when you run the following snippet?

lst = []

del lst

print(lst)

Check

NameError: name 'lst' is not defined

**Exercise 4**

What is the output of the following snippet?

lst = [1, [2, 3], 4]

print(lst[1])

print(len(lst))

Check

[2, 3]

3

**The bubble sort**

Now that you can effectively juggle the elements of lists, it's time to learn how to **sort** them. Many sorting algorithms have been invented so far, which differ a lot in speed, as well as in complexity. We are going to show you a very simple algorithm, easy to understand, but unfortunately not too efficient, either. It's used very rarely, and certainly not for large and extensive lists.

Let's say that a list can be sorted in two ways:

* increasing (or more precisely - non-decreasing) - if in every pair of adjacent elements, the former element is not greater than the latter;
* decreasing (or more precisely - non-increasing) - if in every pair of adjacent elements, the former element is not less than the latter.

In the following sections, we'll sort the list in increasing order, so that the numbers will be ordered from the smallest to the largest.

Here's the list:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 10 | 6 | 2 | 4 |

We'll try to use the following approach: we'll take the first and the second elements and compare them; if we determine that they're in the wrong order (i.e., the first is greater than the second), we'll swap them round; if their order is valid, we'll do nothing. A glance at our list confirms the latter - the elements 01 and 02 are in the proper order, as in 8 < 10.

Now look at the second and the third elements. They're in the wrong positions. We have to swap them:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | **6** | **10** | 2 | 4 |

We go further, and look at the third and the fourth elements. Again, this is not what it's supposed to be like. We have to swap them:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 6 | **2** | **10** | 4 |

Now we check the fourth and the fifth elements. Yes, they too are in the wrong positions. Another swap occurs:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 6 | 2 | **4** | **10** |

The first pass through the list is already finished. We're still far from finishing our job, but something curious has happened in the meantime. The largest element, 10, has already gone to the end of the list. Note that this is the **desired place** for it. All the remaining elements form a picturesque mess, but this one is already in place.

Now, for a moment, try to imagine the list in a slightly different way - namely, like this:

|  |
| --- |
| 10 |
| 4 |
| 2 |
| 6 |
| 8 |

Look - 10 is at the top. We could say that it floated up from the bottom to the surface, just like the **bubble in a glass of champagne**. The sorting method derives its name from the same observation - it's called a **bubble sort**.

Now we start with the second pass through the list. We look at the first and second elements - a swap is necessary:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **6** | **8** | 2 | 4 | 10 |

Time for the second and third elements: we have to swap them too:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | **2** | **8** | 4 | 10 |

Now the third and fourth elements, and the second pass is finished, as 8 is already in place:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 2 | **4** | **8** | 10 |

We start the next pass immediately. Watch the first and the second elements carefully - another swap is needed:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **6** | 4 | 8 | 10 |

Now 6 needs to go into place. We swap the second and the third elements:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | **4** | **6** | 8 | 10 |

The list is already sorted. We have nothing more to do. This is exactly what we want.

As you can see, the essence of this algorithm is simple: **we compare the adjacent elements, and by swapping some of them, we achieve our goal**.

Let's code in Python all the actions performed during a single pass through the list, and then we'll consider how many passes we actually need to perform it. We haven't explained this so far, and we'll do that a little later.

# Sorting a list

How many passes do we need to sort the entire list?

We solve this issue in the following way: **we introduce another variable**; its task is to observe if any swap has been done during the pass or not; if there is no swap, then the list is already sorted, and nothing more has to be done. We create a variable named swapped, and we assign a value of False to it, to indicate that there are no swaps. Otherwise, it will be assigned True.

my\_list = [8, 10, 6, 2, 4] # list to sort

for i in range(len(my\_list) - 1): # we need (5 - 1) comparisons

if my\_list[i] > my\_list[i + 1]: # compare adjacent elements

my\_list[i], my\_list[i + 1] = my\_list[i + 1], my\_list[i] # If we end up here, we have to swap the elements.

You should be able to read and understand this program without any problems:

my\_list = [8, 10, 6, 2, 4] # list to sort

swapped = True # It's a little fake, we need it to enter the while loop.

while swapped:

swapped = False # no swaps so far

for i in range(len(my\_list) - 1):

if my\_list[i] > my\_list[i + 1]:

swapped = True # a swap occurred!

my\_list[i], my\_list[i + 1] = my\_list[i + 1], my\_list[i]

print(my\_list)

Run the program and test it.

# The bubble sort - interactive version

In the editor you can see a complete program, enriched by a conversation with the user, and allowing the user to enter and to print elements from the list: **The bubble sort - final interactive version**.

Python, however, has its own sorting mechanisms. No one needs to write their own sorts, as there is a sufficient number of **ready-to-use tools**.

We explained this sorting system to you because it's important to learn how to process a list's contents, and to show you how real sorting may work.

If you want Python to sort your list, you can do it like this:

my\_list = [8, 10, 6, 2, 4]

my\_list.sort()

print(my\_list)

It is as simple as that.

The snippet's output is as follows:

[2, 4, 6, 8, 10]

**output**

As you can see, all the lists have a method named sort(), which sorts them as fast as possible. You've already learned about some of the list methods before, and you're going to learn more about others very soon.

my\_list = []

swapped = True

num = int(input("How many elements do you want to sort: "))

for i in range(num):

val = float(input("Enter a list element: "))

my\_list.append(val)

while swapped:

swapped = False

for i in range(len(my\_list) - 1):

if my\_list[i] > my\_list[i + 1]:

swapped = True

my\_list[i], my\_list[i + 1] = my\_list[i + 1], my\_list[i]

print("\nSorted:")

print(my\_list)

# Key takeaways

1. You can use the sort() method to sort elements of a list, e.g.:

lst = [5, 3, 1, 2, 4]

print(lst)

lst.sort()

print(lst) # outputs: [1, 2, 3, 4, 5]

2. There is also a list method called reverse(), which you can use to reverse the list, e.g.:

lst = [5, 3, 1, 2, 4]

print(lst)

lst.reverse()

print(lst) # outputs: [4, 2, 1, 3, 5]

**Exercise 1**

What is the output of the following snippet?

lst = ["D", "F", "A", "Z"]

lst.sort()

print(lst)

Check

['A', 'D', 'F', 'Z']

**Exercise 2**

What is the output of the following snippet?

a = 3

b = 1

c = 2

lst = [a, c, b]

lst.sort()

print(lst)

Check

[1, 2, 3]

**Exercise 3**

What is the output of the following snippet?

a = "A"

b = "B"

c = "C"

d = " "

lst = [a, b, c, d]

lst.reverse()

print(lst)

Check

[' ', 'C', 'B', 'A']

**The inner life of lists**

Now we want to show you one important, and very surprising, feature of lists, which strongly distinguishes them from ordinary variables.

We want you to memorize it - it may affect your future programs, and cause severe problems if forgotten or overlooked.

Take a look at the snippet in the editor.

The program:

* creates a one-element list named list\_1;
* assigns it to a new list named list\_2;
* changes the only element of list\_1;
* prints out list\_2.

The surprising part is the fact that the program will output: [2], not [1], which seems to be the obvious solution.

Lists (and many other complex Python entities) are stored in different ways than ordinary (scalar) variables.

You could say that:

* the name of an ordinary variable is the **name of its content**;
* the name of a list is the name of a **memory location where the list is stored**.

Read these two lines once more - the difference is essential for understanding what we are going to talk about next.

The assignment: list\_2 = list\_1 copies the name of the array, not its contents. In effect, the two names (list\_1 and list\_2) identify the same location in the computer memory. Modifying one of them affects the other, and vice versa.

How do you cope with that?

list\_1 = [1]

list\_2 = list\_1

list\_1[0] = 2

print(list\_2)

# Powerful slices

Fortunately, the solution is at your fingertips - its name is the **slice**.

A slice is an element of Python syntax that allows you to **make a brand new copy of a list, or parts of a list**.

It actually copies the list's contents, not the list's name.

This is exactly what you need. Take a look at the snippet below:

list\_1 = [1]

list\_2 = list\_1[:]

list\_1[0] = 2

print(list\_2)

Its output is [1].

This inconspicuous part of the code described as [:] is able to produce a brand new list.

One of the most general forms of the slice looks as follows:

my\_list[start:end]

As you can see, it resembles indexing, but the colon inside makes a big difference.

A slice of this form **makes a new (target) list, taking elements from the source list - the elements of the indices from start to**end - 1.

Note: not to end but to end - 1. An element with an index equal to end is the first element which **does not take part in the slicing**.

Using negative values for both start and end is possible (just like in indexing).

Take a look at the snippet:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[1:3]

print(new\_list)

The new\_list list will have end - start (3 - 1 = 2) elements - the ones with indices equal to 1 and 2 (but not 3).

The snippet's output is: [8, 6]

# Copying the entire list.

list\_1 = [1]

list\_2 = list\_1[:]

list\_1[0] = 2

print(list\_2)

# Copying some part of the list.

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[1:3]

print(new\_list)

**Slices - negative indices**

Look at the snippet below:

my\_list[start:end]

To repeat:

* start is the index of the first element **included in the slice**;
* end is the index of the first element **not included in the slice.**

This is how **negative indices** work with the slice:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[1:-1]

print(new\_list)

The snippet's output is:

[8, 6, 4]

**output**

If the start specifies an element lying further than the one described by the end (from the list's beginning point of view), the slice will be **empty**:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[-1:1]

print(new\_list)

The snippet's output is:

[]

**output**

# Slices: continued

If you omit the start in your slice, it is assumed that you want to get a slice beginning at the element with index 0.

In other words, the slice of this form:

my\_list[:end]

is a more compact equivalent of:

my\_list[0:end]

Look at the snippet below:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[:3]

print(new\_list)

This is why its output is: [10, 8, 6].

Similarly, if you omit the end in your slice, it is assumed that you want the slice to end at the element with the index len(my\_list).

In other words, the slice of this form:

my\_list[start:]

is a more compact equivalent of:

my\_list[start:len(my\_list)]

Look at the following snippet:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[3:]

print(new\_list)

Its output is therefore: [4, 2].

# Slices: continued

As we've said before, omitting both start and end makes a **copy of the whole list**:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[:]

print(new\_list)

The snippet's output is: [10, 8, 6, 4, 2].

The previously described del instruction is able to **delete more than just a list's element at once - it can delete slices too**:

my\_list = [10, 8, 6, 4, 2]

del my\_list[1:3]

print(my\_list)

Note: in this case, the slice **doesn't produce any new list**!

The snippet's output is: [10, 4, 2].

Deleting **all the elements** at once is possible too:

my\_list = [10, 8, 6, 4, 2]

del my\_list[:]

print(my\_list)

The list becomes empty, and the output is: [].

Removing the slice from the code changes its meaning dramatically.

Take a look:

my\_list = [10, 8, 6, 4, 2]

del my\_list

print(my\_list)

The del instruction will **delete the list itself, not its content**.

The print() function invocation from the last line of the code will then cause a runtime error.

# The in and not in operators

Python offers two very powerful operators, able to **look through the list in order to check whether a specific value is stored inside the list or not**.

These operators are:

elem in my\_list

elem not in my\_list

The first of them (in) checks if a given element (its left argument) is currently stored somewhere inside the list (the right argument) - the operator returns True in this case.

The second (not in) checks if a given element (its left argument) is absent in a list - the operator returns True in this case.

Look at the code in the editor. The snippet shows both operators in action. Can you guess its output? Run the program to check if you were right.

my\_list = [0, 3, 12, 8, 2]

print(5 in my\_list)

print(5 not in my\_list)

print(12 in my\_list)

# Lists - some simple programs

Now we want to show you some simple programs utilizing lists.

The first of them tries to find the greater value in the list. Look at the code in the editor.

The concept is rather simple - we temporarily assume that the first element is the largest one, and check the hypothesis against all the remaining elements in the list.

The code outputs 17 (as expected).

The code may be rewritten to make use of the newly introduced form of the for loop:

my\_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]

largest = my\_list[0]

for i in my\_list:

if i > largest:

largest = i

print(largest)

The program above performs one unnecessary comparison, when the first element is compared with itself, but this isn't a problem at all.

The code outputs 17, too (nothing unusual).

If you need to save computer power, you can use a slice:

my\_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]

largest = my\_list[0]

for i in my\_list[1:]:

if i > largest:

largest = i

print(largest)

The question is: which of these two actions consumes more computer resources - just one comparison, or slicing almost all of a list's elements?

my\_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]

largest = my\_list[0]

for i in range(1, len(my\_list)):

if my\_list[i] > largest:

largest = my\_list[i]

print(largest)

**Lists - some simple programs**

Now let's find the location of a given element inside a list:

my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

to\_find = 5

found = False

for i in range(len(my\_list)):

found = my\_list[i] == to\_find

if found:

break

if found:

print("Element found at index", i)

else:

print("absent")

Note:

* the target value is stored in the to\_find variable;
* the current status of the search is stored in the found variable (True/False)
* when found becomes True, the for loop is exited.

Let's assume that you've chosen the following numbers in the lottery: 3, 7, 11, 42, 34, 49.

The numbers that have been drawn are: 5, 11, 9, 42, 3, 49.

The question is: how many numbers have you hit?

The program will give you the answer:

drawn = [5, 11, 9, 42, 3, 49]

bets = [3, 7, 11, 42, 34, 49]

hits = 0

for number in bets:

if number in drawn:

hits += 1

print(hits)

Note:

* the drawn list stores all the drawn numbers;
* the bets list stores your bets;
* the hits variable counts your hits.

The program output is: 4.

**LAB**

## Estimated time

10-15 minutes

## Level of difficulty

Easy

## Objectives

Familiarize the student with:

* list indexing;
* utilizing the in and not in operators.

## Scenario

Imagine a list - not very long, not very complicated, just a simple list containing some integer numbers. Some of these numbers may be repeated, and this is the clue. We don't want any repetitions. We want them to be removed.

Your task is to write a program which removes all the number repetitions from the list. The goal is to have a list in which all the numbers appear not more than once.

Note: assume that the source list is hard-coded inside the code - you don't have to enter it from the keyboard. Of course, you can improve the code and add a part that can carry out a conversation with the user and obtain all the data from her/him.

Hint: we encourage you to create a new list as a temporary work area - you don't need to update the list in situ.

We've provided no test data, as that would be too easy. You can use our skeleton instead.

my\_list = [1, 2, 4, 4, 1, 4, 2, 6, 2, 9]

#

# Write your code here.

#

print("The list with unique elements only:")

print(my\_list)

# Key takeaways

1. If you have a list l1, then the following assignment: l2 = l1 does not make a copy of the l1 list, but makes the variables l1 and l2 **point to one and the same list in memory**. For example:

vehicles\_one = ['car', 'bicycle', 'motor']

print(vehicles\_one) # outputs: ['car', 'bicycle', 'motor']

vehicles\_two = vehicles\_one

del vehicles\_one[0] # deletes 'car'

print(vehicles\_two) # outputs: ['bicycle', 'motor']

2. If you want to copy a list or part of the list, you can do it by performing **slicing**:

colors = ['red', 'green', 'orange']

copy\_whole\_colors = colors[:] # copy the entire list

copy\_part\_colors = colors[0:2] # copy part of the list

3. You can use **negative indices** to perform slices, too. For example:

sample\_list = ["A", "B", "C", "D", "E"]

new\_list = sample\_list[2:-1]

print(new\_list) # outputs: ['C', 'D']

4. The start and end parameters are **optional** when performing a slice: list[start:end], e.g.:

my\_list = [1, 2, 3, 4, 5]

slice\_one = my\_list[2: ]

slice\_two = my\_list[ :2]

slice\_three = my\_list[-2: ]

print(slice\_one) # outputs: [3, 4, 5]

print(slice\_two) # outputs: [1, 2]

print(slice\_three) # outputs: [4, 5]

5. You can **delete slices** using the del instruction:

my\_list = [1, 2, 3, 4, 5]

del my\_list[0:2]

print(my\_list) # outputs: [3, 4, 5]

del my\_list[:]

print(my\_list) # deletes the list content, outputs: []

6. You can test if some items **exist in a list or not** using the keywords in and not in, e.g.:

my\_list = ["A", "B", 1, 2]

print("A" in my\_list) # outputs: True

print("C" not in my\_list) # outputs: True

print(2 not in my\_list) # outputs: False

**Exercise 1**

What is the output of the following snippet?

list\_1 = ["A", "B", "C"]

list\_2 = list\_1

list\_3 = list\_2

del list\_1[0]

del list\_2[0]

print(list\_3)

Check

['C']

**Exercise 2**

What is the output of the following snippet?

list\_1 = ["A", "B", "C"]

list\_2 = list\_1

list\_3 = list\_2

del list\_1[0]

del list\_2

print(list\_3)

Check

['B', 'C']

**Exercise 3**

What is the output of the following snippet?

list\_1 = ["A", "B", "C"]

list\_2 = list\_1

list\_3 = list\_2

del list\_1[0]

del list\_2[:]

print(list\_3)

Check

[]

**Exercise 4**

What is the output of the following snippet?

list\_1 = ["A", "B", "C"]

list\_2 = list\_1[:]

list\_3 = list\_2[:]

del list\_1[0]

del list\_2[0]

print(list\_3)  
Check

['A', 'B', 'C']

**Exercise 5**

Insert in or not in instead of ??? so that the code outputs the expected result.

my\_list = [1, 2, "in", True, "ABC"]

print(1 ??? my\_list) # outputs True

print("A" ??? my\_list) # outputs True

print(3 ??? my\_list) # outputs True

print(False ??? my\_list) # outputs False  
Check

my\_list = [1, 2, "in", True, "ABC"]

print(1 in my\_list) # outputs True

print("A" not in my\_list) # outputs True

print(3 not in my\_list) # outputs True

print(False in my\_list) # outputs False

**Lists in lists**

Lists can consist of scalars (namely numbers) and elements of a much more complex structure (you've already seen such examples as strings, booleans, or even other lists in the previous Section Summary lessons). Let's have a closer look at the case where a **list's elements are just lists**.

We often find such **arrays** in our lives. Probably the best example of this is a **chessboard**.

A chessboard is composed of rows and columns. There are eight rows and eight columns. Each column is marked with the letters A through H. Each line is marked with a number from one to eight.

The location of each field is identified by letter-digit pairs. Thus, we know that the bottom left corner of the board (the one with the white rook) is A1, while the opposite corner is H8.

Let's assume that we're able to use the selected numbers to represent any chess piece. We can also assume that **every row on the chessboard is a list**.

Look at the code below:

row = []

for i in range(8):

row.append(WHITE\_PAWN)

It builds a list containing eight elements representing the second row of the chessboard - the one filled with pawns (assume that WHITE\_PAWN is a **predefined symbol** representing a white pawn).

The same effect may be achieved by means of a **list comprehension**, the special syntax used by Python in order to fill massive lists.

A list comprehension is actually a list, but **created on-the-fly during program execution, and is not described statically**.

Take a look at the snippet:

row = [WHITE\_PAWN for i in range(8)]

The part of the code placed inside the brackets specifies:

* the data to be used to fill the list (WHITE\_PAWN)
* the clause specifying how many times the data occurs inside the list (for i in range(8))

Let us show you some other **list comprehension examples**:

Example #1:

squares = [x \*\* 2 for x in range(10)]

The snippet produces a ten-element list filled with squares of ten integer numbers starting from zero (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

Example #2:

twos = [2 \*\* i for i in range(8)]

The snippet creates an eight-element array containing the first eight powers of two (1, 2, 4, 8, 16, 32, 64, 128)

Example #3:

odds = [x for x in squares if x % 2 != 0 ]

The snippet makes a list with only the odd elements of the squares list.

**Lists in lists: two-dimensional arrays**

Let's also assume that a **predefined symbol** named EMPTY designates an empty field on the chessboard.

So, if we want to create a list of lists representing the whole chessboard, it may be done in the following way:

board = []

for i in range(8):

row = [EMPTY for i in range(8)]

board.append(row)

Note:

* the inner part of the loop creates a row consisting of eight elements (each of them equal to EMPTY) and appends it to the board list;
* the outer part repeats it eight times;
* in total, the board list consists of 64 elements (all equal to EMPTY)

This model perfectly mimics the real chessboard, which is in fact an eight-element list of elements, all being single rows. Let's summarize our observations:

* the elements of the rows are fields, eight of them per row;
* the elements of the chessboard are rows, eight of them per chessboard.

The board variable is now a **two-dimensional array**. It's also called, by analogy to algebraic terms, a **matrix**.

As list comprehensions can be **nested**, we can shorten the board creation in the following way:

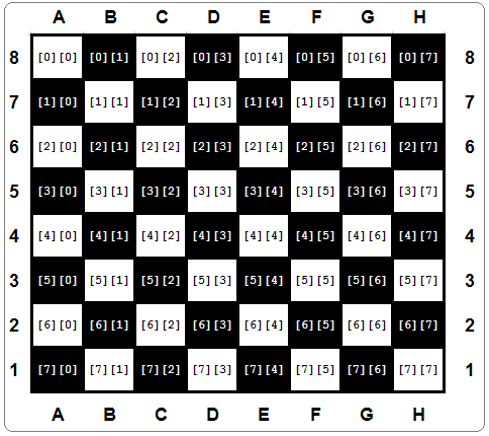
board = [[EMPTY for i in range(8)] for j in range(8)]

The inner part creates a row, and the outer part builds a list of rows.

# Lists in lists: two-dimensional arrays - continued

Access to the selected field of the board requires two indices - the first selects the row; the second - the field number inside the row, which is de facto a column number.

Take a look at the chessboard. Every field contains a pair of indices which should be given to access the field's content:



Glancing at the figure shown above, let's set some chess pieces on the board. First, let's add all the rooks:

board[0][0] = ROOK

board[0][7] = ROOK

board[7][0] = ROOK

board[7][7] = ROOK

If you want to add a knight to C4, you do it as follows:

board[4][2] = KNIGHT

And now a pawn to E5:

board[3][4] = PAWN

And now - experiment with the code in the editor.

EMPTY = "-"

ROOK = "ROOK"

board = []

for i in range(8):

row = [EMPTY for i in range(8)]

board.append(row)

board[0][0] = ROOK

board[0][7] = ROOK

board[7][0] = ROOK

board[7][7] = ROOK

print(board)

**Multidimensional nature of lists: advanced applications**

Let's go deeper into the multidimensional nature of lists. To find any element of a two-dimensional list, you have to use two *coordinates*:

* a vertical one (row number)
* and a horizontal one (column number).

Imagine that you develop a piece of software for an automatic weather station. The device records the air temperature on an hourly basis and does it throughout the month. This gives you a total of 24 × 31 = 744 values. Let's try to design a list capable of storing all these results.

First, you have to decide which data type would be adequate for this application. In this case, a float would be best, since this thermometer is able to measure the temperature with an accuracy of 0.1 ℃.

Then you take an arbitrary decision that the rows will record the readings every hour on the hour (so the row will have 24 elements) and each of the rows will be assigned to one day of the month (let's assume that each month has 31 days, so you need 31 rows). Here's the appropriate pair of comprehensions (h is for hour, d for day):

temps = [[0.0 for h in range(24)] for d in range(31)]

The whole matrix is filled with zeros now. You can assume that it's updated automatically using special hardware agents. The thing you have to do is to wait for the matrix to be filled with measurements.

Now it's time to determine the monthly average noon temperature. Add up all 31 readings recorded at noon and divide the sum by 31. You can assume that the midnight temperature is stored first. Here's the relevant code:

temps = [[0.0 for h in range(24)] for d in range(31)]

#

# The matrix is magically updated here.

#

total = 0.0

for day in temps:

total += day[11]

average = total / 31

print("Average temperature at noon:", average)

Note: the day variable used by the for loop is not a scalar - each pass through the temps matrix assigns it with the subsequent rows of the matrix; hence, it's a list. It has to be indexed with 11 to access the temperature value measured at noon.

Now find the highest temperature during the whole month - see the code:

temps = [[0.0 for h in range(24)] for d in range(31)]

#

# The matrix is magically updated here.

#

highest = -100.0

for day in temps:

for temp in day:

if temp > highest:

highest = temp

print("The highest temperature was:", highest)

Note:

* the day variable iterates through all the rows in the temps matrix;
* the temp variable iterates through all the measurements taken in one day.

Now count the days when the temperature at noon was at least 20 ℃:

temps = [[0.0 for h in range(24)] for d in range(31)]

#

# The matrix is magically updated here.

#

hot\_days = 0

for day in temps:

if day[11] > 20.0:

hot\_days += 1

print(hot\_days, "days were hot.")

# Three-dimensional arrays

Python does not limit the depth of list-in-list inclusion. Here you can see an example of a three-dimensional array:

Imagine a hotel. It's a huge hotel consisting of three buildings, 15 floors each. There are 20 rooms on each floor. For this, you need an array which can collect and process information on the occupied/free rooms.

First step - the type of the array's elements. In this case, a Boolean value (True/False) would fit.

Step two - calm analysis of the situation. Summarize the available information: three buildings, 15 floors, 20 rooms.

Now you can create the array:

rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]

The first index (0 through 2) selects one of the buildings; the second (0 through 14) selects the floor, the third (0 through 19) selects the room number. All rooms are initially free.

Now you can book a room for two newlyweds: in the second building, on the tenth floor, room 14:

rooms[1][9][13] = True

and release the second room on the fifth floor located in the first building:

rooms[0][4][1] = False

Check if there are any vacancies on the 15th floor of the third building:

vacancy = 0

for room\_number in range(20):

if not rooms[2][14][room\_number]:

vacancy += 1

The vacancy variable contains 0 if all the rooms are occupied, or the number of available rooms otherwise.

Congratulations! You've made it to the end of the module. Keep up the good work!

rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]

# Key takeaways

1. **List comprehension** allows you to create new lists from existing ones in a concise and elegant way. The syntax of a list comprehension looks as follows:

[expression for element in list if conditional]

which is actually an equivalent of the following code:

for element in list:

if conditional:

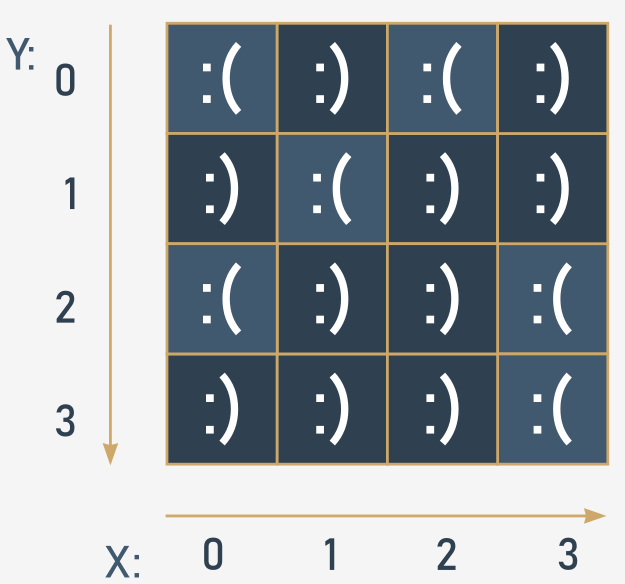
expression

Here's an example of a list comprehension ‒ the code creates a five-element list filled with the first five natural numbers raised to the power of 3:

cubed = [num \*\* 3 for num in range(5)]

print(cubed) # outputs: [0, 1, 8, 27, 64]

2. You can use **nested lists** in Python to create **matrices** (i.e., two-dimensional lists). For example:



# A four-column/four-row table ‒ a two dimensional array (4x4)

table = [[":(", ":)", ":(", ":)"],

[":)", ":(", ":)", ":)"],

[":(", ":)", ":)", ":("],

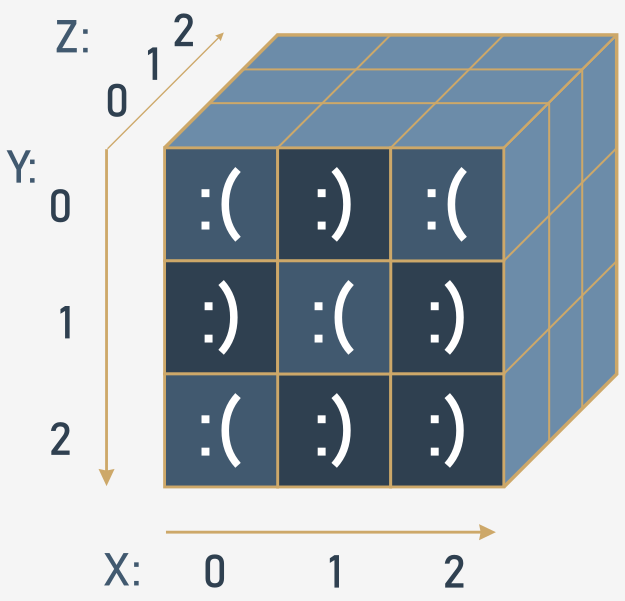
[":)", ":)", ":)", ":("]]

print(table)

print(table[0][0]) # outputs: ':('

print(table[0][3]) # outputs: ':)'

3. You can nest as many lists in lists as you want, thereby creating n-dimensional lists, e.g., three-, four- or even sixty-four-dimensional arrays. For example:



# Cube - a three-dimensional array (3x3x3)

cube = [[[':(', 'x', 'x'],

[':)', 'x', 'x'],

[':(', 'x', 'x']],

[[':)', 'x', 'x'],

[':(', 'x', 'x'],

[':)', 'x', 'x']],

[[':(', 'x', 'x'],

[':)', 'x', 'x'],

[':)', 'x', 'x']]]

print(cube)

print(cube[0][0][0]) # outputs: ':('

print(cube[2][2][0]) # outputs: ':)'

**Congratulations! You have completed Module 3.**

Well done! You've reached the end of Module 3 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 3:

* Boolean values to compare different values and control the execution paths using the if and if-else instructions;
* the utilization of loops (while and for) and how to control their behavior using the break and continue instructions;
* the difference between logical and bitwise operations;
* the concept of lists and list processing, including the iteration provided by the for loop, and slicing;
* the idea of multi-dimensional arrays.

You are now ready to take the module quiz and attempt the final challenge: Module 3 Test, which will help you gauge what you've learned so far.