Python tutorial

Obsah

Hello, you've come across a Python tutorial written on the occasion of the Online School of Programming 2020. Its authors are Roman and Andrej (Andrej and Roman). We are very grateful to Žaba for proofreading and structuring the tutorial. If you find any errors or passages that you think we didn't explain well, feel free to contact us. How can you recognize a poorly written passage? For example, if even after reading it several times, you don't understand what the text or code means. You can contact us at andrejkorman@gmail.com and r.sobkuliak@gmail.com.

1 Introduction

Before we start with Python, it is important to answer a few questions. The first and most important one is, what is programming? Programming is creating programs that we can run on a computer. An example of a program could be an e-book reader, a web browser, or something simpler, like a calculator. The important thing about programs is that they have input and output. For a program functioning as a calculator, the input can be a mathematical expression like $5 + 3 \cdot 7$ and the output the value of this expression, which is 26. In this tutorial, we will focus on simple programs. In the case of a calculator on your computer, you are probably used to entering input, a mathematical expression, using buttons, i.e., some graphical interface (buttons, windows, etc.). Our programs will not have such an interface, which, by the way, makes them considerably shorter. Both input and output will be in textual form.

Now that we know what a program is, we need to know how to create one. This is where programming languages come in, as you might guess. They allow us to create code, a sequence of commands in a given language, which together forms a program. After running, a program receives some input from us and produces an answer (output) that it typically displays in some way, for example, by showing it on the display.

1.1 Why Python?

There are countless programming languages. Why did we choose Python in this text? We think its commands are natural, and at the same time, it is widely used in practice. According to the StackOverflow survey from 2019, it is the second most popular programming language. So, Python will open the doors to many corners of computer science for you :)

1.2 Installation

Before you write your first program, you need to install Python. There are many advanced tools for programming in Python. However, we won't use them for now because they are unnecessarily complicated for beginners. Instead, we'll use a simple editor called `Python IDLE`, created by the author of Python himself. This editor supports all standard operating systems (Windows, MacOS, Linux), and you can download it from the official Python website. We recommend installing the latest version, which is version 3.8.2 at the time of writing. Alternatively, you can use an online interpreter such as Google Colab.

2 First Programs and `print`

Once we have Python installed, let's open the Python IDLE program. This is the editor in which we will write our programs. Choose `File -> New File` and copy the following program into the new window:

```
# First program in Python
print(5)
print(5+2+7)
print(7-10)
print(2*15)
print(15/10)
print(15//10)
print(7%3)
```

Then save the program using `File -> Save` in any folder. We recommend creating a separate folder to save your programs. Now you can run the program using `Run -> Run Module` or by pressing `F5`. If you're using Google Colab, create a new Code block by clicking on `+Code`, paste the above code, and run it using the `Play symbol` on the left of the block.

What you see above is a code (sequence of commands) forming a very simple program. The first line is not essential, we will stop for a bit to explain its purpose. Sometimes, when someone reads your code, you want to leave a bit of extra information. It might explain what a more complicated part of the code does or highlight

certain commands. However, in the code, everything is considered a command. Therefore, there are so-called comments - parts of the text that are not executed and serve for the needs of us humans. The character # is used to leave a comment. Everything we write after it is considered only a comment. Python ignores these comments when executing individual commands. Now let's get to the essentials.

In this program, we use the `print` function. A function is a smaller sequence of commands that repeats or is logically cohesive enough to be named. Essentially, it is a small program within a program. We usually give several *parameters* to a function, telling it what to perform the operation on. We can probably guess what this particular function does - it prints what is inside the parentheses. This is all you need to know about functions for now. Functions are a relatively complex topic, which is further addressed in Chapter 10.

Above, we see that `print` is a function with a parameter of what we want to print. Run the program and look at its output. Consider which lines surprised you. It is logical that the expression 15/10 did not evaluate to a whole but a decimal number. However, special attention should be given to the expression 15//10. Two division signs in a row in Python often indicate the so-called integer division, i.e., division without a remainder. $x//y$ can be interpreted as „How many times does y fit fully into x?“ In other words, it is a division with everything after the decimal point omitted. You may find the expression 7%3 new, where % indicates the remainder after integer division. The result of this operation is the part of the number that „remains“ after integer division. $7\%3 = 1$ because $7//3 = 2$, and what remains can be expressed as $7 - 2 \cdot 3 = 1$.

In this program, we have shown that `print` can print numbers. All expressions ($7 - 10$, $5 + 2 + 7$, etc.) containing basic mathematical operations are directly evaluated to numbers (-3, 14), and the task of `print` is only to print the given number. However, `print` can do more. Copy the following code into your code file:

```
print("Python tutorial!")
print("Python" + "tutorial!")
print(2 + 3)
print("2" + "3")
```

Review the code again, run it, and compare your expectations with the results on the output. On the first line, we see that we can print text. Such text in Python is called a *string of characters* or simply a *string*. It consists of characters enclosed in quotes, either `"<text>"` or `'<text>'`. Note that it doesn't matter whether we use „double quotes“ or an apostrophe. Most often, when we talk about characters in Python, we mean the English alphabet, digits, punctuation, parentheses, and so on. In this text, we don't work with accents, soft diacritics, umlauts, and other non-English alphabet deviations at all.

In the second line, we see that, like numbers, we can concatenate strings. The plus sign in this case concatenates the strings together. How is it possible that the result of the 3rd and 4th lines differs? The reason is that in the first case, it's a plus applied to numbers. In the second case, we applied plus to strings. Python sees that in the 4th line, on both sides of the plus sign, there are numbers with quotes around them, indicating that we want to represent these numbers as strings. Therefore, Python has no choice but to use plus for strings. When we try to combine these two representations and write

```
print(2 + "3")
# Error!
# TypeError: unsupported operand type(s) for +: 'int' and 'str'
# This program will end with the error above. Python tells us in this
# error that the + operator cannot be used between 'int' and 'str' types.
```

we will cause an error. This is because Python does not understand how to add *int* (integer) and *str* (string). However, this doesn't apply if we replace the plus sign with the multiplication sign. Try and guess, however, what will the following program print? :) Afterwards, run it to test your hypothesis.

```
print(2 * "Hello!")
```

The last thing regarding the `print` statement is that every `print` statement writes its output on its own line. Now we'll reveal something to you. The fact that text goes to the next line after each `print` is due to the presence of a character in the text, which is not among the so-called *printable characters*. It is not visible when printed, but it causes the text after it to be on a new line. This character can be written as `\n`. You can try it by adding the following command to your program:

```
print("Here will be a new line->\n and here->\nwas a new line.")
# Output:
# Here will be a new line->
#  and here->
# was a new line.
```

Therefore, the `print` function does nothing more than add `\n` to the end of what we want to print. The creators of the `print` function thought that perhaps we don't want to put the end-of-line character behind what we print and added the possibility to change the end of the printed text using a parameter. This parameter is

additionally *named*, and it is probably not surprising that it is called `end`. Therefore, if instead of a newline character, we want to have two `X` characters at the end, we will do it as follows.

```
print("We␣stay", end="XX")
print("on␣the␣same␣line.")
# We stayXXon the same line.
```

Especially if we don't want to print anything after `print`, we give an empty string to the `end` parameter.

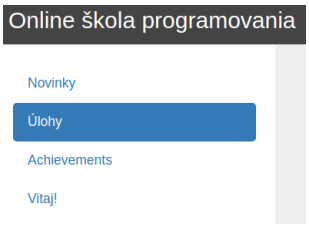
```
print("Staying␣", end='')
print("on␣the␣same␣line.")
# Staying on the same line.
```

3 Examples

At the end of each chapter, you will find a set of exercises that you should solve before continuing. The exercises are designed to test your skills and generally increase in difficulty. In this chapter, we have prepared only one task for you: Hello

However, you have probably never submitted a program to our tester, so let us briefly explain the whole process. The tester is our server, where you send your program that solves a particular task. Our tester runs your program on various inputs and checks if it does what is required in the assignment. Based on that, it evaluates it as OK, WA (Wrong Answer), or something else. If you're interested, see the main page for all possible results. Don't worry, you can submit as many times as you want. However, we don't recommend getting stuck and spending hours on the same problem. If you are struggling and getting frustrated, ask for help.

Now, let's talk about how to submit your program to the judge. You can find the judge at zurich.ksp.sk. For the first time, choose the option `Register` and register as you would on any other website. After logging in, you can navigate through the page using the left panel. When you decide to solve a problem, go to the `Tasks` section through the left panel.



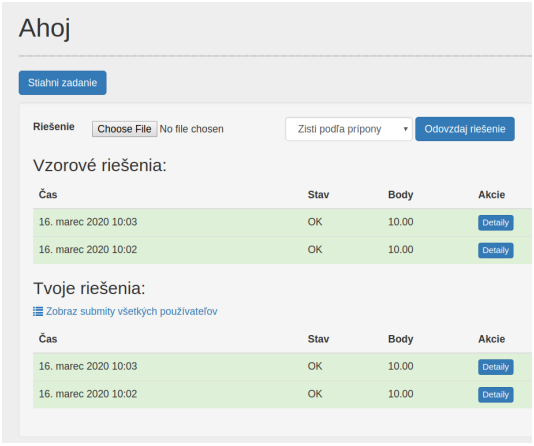
Then notice a set of examples called `Introduction`, and click on `Show Problems` to reveal the tasks:

After clicking on an example, you will be taken to the page that the link of the task in this tutorial would otherwise lead you to. You can submit your code to a task by copy-pasting your code into the input field and click `Submit Solution`.

Your submissions (attempts to solve) are at the bottom, where a correct solution has an OK status. After solving the task, you can also click on `Sample Solutions`, which are usually available in two or three languages.

4 Variables

A program often needs to remember some information for its operation, whether it's user input or intermediate results of the calculation the program is performing. In Python, variables are used for this purpose. A variable is like a box (a place in the computer's memory) where the program stores data. Additionally, this box is named. Once a box with a name, such as x, is created, we can store a number in it using `=`.




```
x = 4   # store the number 4 in the box x
```

Later on, we can use the value in this box like this:

```
print(5*x)   # when evaluating the expression, x is replaced by 4
```

We can put a value into the box and use it repeatedly.

```
a = 10
print(4*a)   # prints: 40
a = 7
print(4*a)   # prints: 28
```

So far, we've concealed how such a variable is created. The secret is that Python automatically creates a variable when we assign it a value for the first time. To create a variable and assign a value to it, all we need to write is:

```
a = 4   # assign 4 to a, the variable a is created
b = 2 * a - 3   # assigns the value of the expression on the right, which is 5, to b
c, d = 10, 12   # we can do multiple assignments at once (c = 10 and d = 12)
a = 8 # we changed the value of a, the variable is not created again
```

At the beginning, we mentioned that a program stores data in a variable, but so far, we've only shown how to store a number in it. Depending on the type of data a variable is currently storing, we refer to it as its `type`. We've encountered three types so far. The first is an integer or `int`. The second is a floating-point number or `float`. The third is a string or `string`. The following table shows the basic types in Python.

Type	Description	Examples
int	integers	1, -10, 0, 123
float	floating-point numbers	0.5, .3, 3.14
bool	True/False	`True`/`False`
string	joined characters	„abc“, 'abc'
None	none of the above types	`None`

Another important concept is the notion of an `operator`. We've already talked about variables and how they have different types and can be used in more complex expressions. By expression, we mean something like $5 \times x + 20 + y$. This expression is evaluated, and its result is returned to us as a number. We can write something like $result = 5 \times x + 20 + y$. In this expression, we use the operator + twice. The + operator takes what's on the left and right, adds it, and returns the result. Of course, Python has more operators. Some you already know, like $-$, //, $*$... We also know that the same operator can behave differently when it has different types on the right and left, returning different types.

```
a, b = 2, 12 # assign 2 to a and 12 to b
c = a + b # c will be 14, + returns an integer here
r1 = "Yes"
r2 = "No"
r3 = r1 + r2 # r3 will be "YesNo", because + returns a string here
r4 = a * r1 # r4 will be "YesYes", because * returns a string here
```

A crucial type in programming is the `bool` type. It can represent only two values: `True` and `False`. The importance of this type becomes apparent in the chapter on conditions. Here's an example of usage:

```
a = True
b = False
print(a)
print(b)
```

Operators we've discussed so far return either integers, floating-point numbers, or strings. Important operators that return `bool` are == and ! =. The first works as a comparison, returning `True` when the thing on the right and left are equal. The second is its exact opposite, returning `True` when the thing on the right and left are not equal. Examples of usage:

```
a, b = 5, 10
c = (a==b) # c is False
d = (a==5) # d is True
e = (a!=b) # e is True
# for readability, we put expressions to the right of = into parentheses
```

Other useful operators that return `bool` are <, >, >=, and <=. Example of usage:

```
a = 5>3 # a is True
b = 7<=8 # b is True
```

5 Input and Output (I/O)

To read input, we use the `input` function. After using it, the program reads one line from the input and returns it to us as a *string*. Naturally, we need to store this string somewhere. It's essential to note that the input read is a string. If we want to work with it as a number, we need to *convert* it to an integer using the `int` function.

```
a = input()  # reads one line from the input and stores it in variable 'a' as a string
print(a + 10)  # Error! Variable 'a' contains a string, and 10 is an int.
```

In the previous example, we see incorrect usage of the `input` function. If you remember functions and their parameters, you can see that we didn't provide any parameters to this function (there is nothing in its parentheses). What we do on the first line is correct, but the error is manifested in the second line, which we didn't intend to make. We wanted to read a number into a, but, in reality, we only read a string into it. To work with variable a as a number, we need to convert what the `input` function returned, a string, into an integer type. We'll use the `int` function for that.

```
a = int(input())  # to create an 'int' type variable 'a', we need to convert the string
print(a + 10)  # Ok, both the variable and the value we add are of 'int' type.
```

For output, we again use the `print` function. We can provide variables/values to it as parameters.

```
# Trick: assigning multiple variables at once. The syntax is equivalent to:
# a = 10
# b = 15
a, b = 10, 15
# print with multiple parameters prints these parameters separated by space
# Because we set end='', there won't be a newline character
print(a, b, end='')
print(a)
# Output:
# 10 1510
# Explanation:
# The first print prints "10 15" (without a newline character)
# and then the second print adds 10 (with a newline character).
```

Taskset

Number, Swap, Rectangle 1

6 Conditions

Often, a program needs to make some decisions. A program is something that behaves the same way each time it is run, but not on every input. This means that the same input will always result in the same program behavior. However, since we often give different inputs to the program, it needs to react to them in some way. For example, we might want the code to execute only if certain conditions are met. A typical example is a program that prints whether a number is or isn't 0. The suitable construct for achieving this kind of program behavior is the `if` statement (if). It simplifies to `if <condition>: <command>`. It means that if the `<condition>` is true, execute the `<command>`. The `<condition>` in this case is any expression that evaluates to `bool` (either `True` or `False`). **Don't forget** that comparison of two things is done using **two** equals signs `(==)`, not one `(=)`.

```
x = int(input())
if x == 0:
    print("It␣is␣zero!")
if x != 0:
    print("It␣is␣not␣zero!")
```

In the previous example, you can also observe another essential thing: the indentation of commands. We are referring to the visible space in the 3rd and 5th lines. Indentation tells Python that this command belongs to the preceding condition. It's crucial if, for example, we want to execute more than one command when the first condition is satisfied. In that case, we can add another indented command:

```
# The program determines if the input is 0.
x = int(input())
if x == 0:
    print("It␣is␣zero!")
    print("I␣will␣print␣this␣too!")

# Input1:
# 0
# Output1:
# It is zero!
# I will print this too!

# Input2:
# 3
# Output2:
# (empty)
```

If we didn't indent the second command, `I will print this too!` would be printed even for a non-zero input. We can achieve indentation by pressing the Tab key.

```
# The program determines if the input is 0.
x = int(input())
if x == 0:
    print("It␣is␣zero!")
print("I␣will␣print␣this␣too!")

# Input1:
# 0
# Output1:
# It is zero!
# I will print this too!

# Input2:
# 3
# Output2:
# I will print this too!
```

Sometimes, there are precisely two possibilities for what we want to do. In that case, the `if <condition>: <command> else: <command>` construct is helpful. That is, if the condition is true, execute the first command; if not, execute the second.

```
x = int(input())
if x == 0:
    print("It␣is␣zero!")
else:
    print("It␣is␣not␣zero!")
```

The last option is when we have more than two possibilities. In that case, we use the `if <condition>: <command> elif <condition>: <command> else: <command>` construct. This option is suitable when there are 2 or more possibilities, and we want **only one** of them to be executed. The number of `elif` statements is arbitrary, but for readability, it's generally recommended not to have more than 3-4.

```
# If two branches are not enough (if, else), we can add more using 'elif'.
if x % 2 == 0:
  print('x␣is␣even')
elif x % 3 == 0:
  print('x␣is␣divisible␣by␣3')
else:
  print('x␣is␣neither␣divisible␣by␣2␣nor␣3')
```

Again, notice that each command after a colon is on a separate line and is indented from the condition.

The condition is any expression that we can evaluate as `True` or `False`. We can also combine multiple expressions using logical operators `and` and `or`. The expression `<condition1> and <condition2>` evaluates to `True` if both conditions are true. The expression `<condition1> or <condition2>` evaluates to `True` if at least one of the conditions is true. In conditions, we can use parentheses to increase the readability of the order of operations.

```
if (x % 2 == 0) and (x % 3 == 0):
  print('x␣is␣divisible␣by␣6')
```

```
# The condition above is equivalent to nested conditions
if x % 2 == 0:
  if x % 3 == 0:
    print('x is divisible by 6')

if (name == 'John') or (name == 'Maria'):
  print('Name is John or Maria')
```

Taskset

Equal, Sign, Smallest, The Middle Man, Average, Remainder, Not the Largest

7 Loops

In a program, we often need to repeat some commands multiple times. For example, if we need to print numbers from 1 to 5, with the current knowledge, we would create the following program:

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

7.1 While Loop

As you might expect, there is a more elegant way to achieve this. We will use the `while` loop.

```
# Print numbers from 1 to 5.
i = 1
while i <= 5:
  print(i)
  i += 1
```

The `while` statement repeats the execution of indented commands *while* the condition specified after the `while` keyword is true. Let's describe this behavior using the provided program. The program first assigns 1 to the variable `i`. Then, the condition `while i <= 5` is evaluated, which is true because `i` has the value of 1. Therefore, the command `print(i)` is executed, and the variable `i` is incremented by 1 (`i += 1`). The condition `while i <= 5` is evaluated again, still true because `i` now has the value of 2. This process continues until the variable `i` reaches the value of 6. At that point, the condition `i <= 5` is false, and the indented commands below the `while` loop are skipped, allowing the program to proceed past the loop.

7.2 For Loop

Often, we know the exact number of loop iterations. This was the case with printing numbers from 1 to 5 (5 iterations). In such cases, it is more elegant to use the `for` loop.

```
# Print numbers from 1 to 5.
for i in range(1, 6):
  print(i)
```

After the `for` keyword, we have the variable name (in this case, i), which will sequentially take on values at each iteration. After the variable name, we have the `in` keyword, followed by the expression `range(1, 6)`. This ensures that the variable i will take on values in the range from 1 to 5.

Now we understand the `for`-loop header. The actual execution works similarly to the `while` loop. The variable `i` is first assigned the value 1 (the first parameter of `range`), the body is executed (`print(i)`), the value of `i` is increased by 1, the body is executed again, and so on until `i` reaches the value of 6 (the second parameter of `range`). For this value, the body is no longer executed.

7.3 Advanced Loops

What if we wanted to create a loop that skips every other number (1, 3, 5, ...)? Or a loop that counts backward (10, 9, 8, ...)? Let's explore how to achieve this.

The `range` function has 3 variants that allow us to create advanced loops:

- `range(stop)` - creates a range with elements from 0 to $stop - 1$ (inclusive).

- `range(start, stop)` - creates a range with elements from *start* to *stop* − 1 (inclusive).
- `range(start, stop, step)` - creates a range with elements from *start* to *stop* − 1 (inclusive), including every *step*-th number.

Usage:

```
# Print numbers from 0 to 7.
for i in range(8):
  print(i)

# Print numbers from -2 to 2.
for i in range(-2, 3):
  print(i)

# Print odd numbers up to 9.
for i in range(1, 10, 2):
  print(i)

# Print numbers from 10 to 1 (backwards).
# Note that the last parameter is negative, so the numbers
# will decrease from start to end. Additionally, the second argument
# (end of the loop) is 0, not 1, because range accepts the number
# immediately after the intended last value.
for i in range(10, 0, -1):
  print(i)
```

Sometimes we need to exit a loop early or skip some values. For this, we use the `continue` and `break` statements. The `break` statement exits the *currently* executing loop. The program then continues with the commands after the loop. For example, the program below stops at number 4.

```
# Print numbers from 0 to 3.
for i in range(11):
  if i == 4:
    break
  print(i)
# Output: 0, 1, 2, 3
# The number 4 will not be printed.
```

The `continue` statement skips to the next iteration of the loop. The remaining part after the `continue` statement is not executed. For example, in the following program, the message `'Is odd!'` is skipped for odd numbers.

```
# Print numbers from 0 to 6.
for i in range(7):
  print(i)
  # Remainder after division is 1, so the number is odd.
  if i % 2 == 1:
    continue
  print('␣Is␣odd!')
# Output:
# 0
# Is odd!
# 1
# 2
# Is odd!
# 3
# (etc.)
```

Taskset

Cycles, The Stripe, Rectangle 2, Triangle, Pyramid, The Largest, Fibonacci

8 List

Imagine that instead of receiving a few numbers as input, we get a dozen of them. Where would we store these numbers? We could create many variables and load the input into them. However, this would not be very elegant; moreover, what if the number of input numbers depends on the previous input? (For example, the first number of the input would determine how many numbers will follow.)

For such a case, we need to expand our arsenal to include `list`. A list declares a sequence of variables stored one after another, which can be accessed by their position.

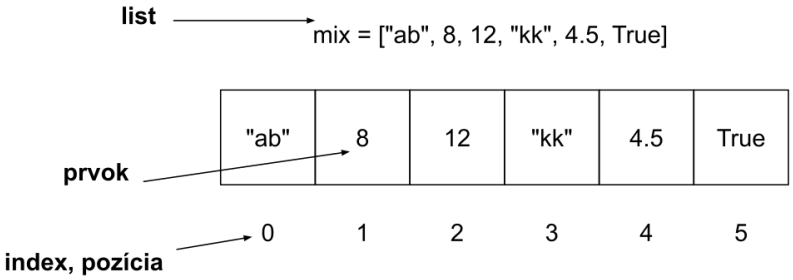
```
# List containing numbers 1 to 5
numbers = [1, 2, 3, 4, 5]

# Printing the first element of the "numbers" list.
print(numbers[0])
# Output: 1
```

```
# Error!
print(numbers[5])
# IndexError: list index out of range
# Python tells us that index 5 is out of the allowed range
# The "numbers" list has 5 elements indexed from 0, so the last valid index is 4, index 5 is invalid.
```

As shown in the example above, creating a list has the format `[<val1>, <val2>, ...]`, where the value can be any variable or value (*literal*). Lists are numbered from 0. So, if we create a string of length 5 as in the example above, the valid indices (positions) are 0 to 4.

You can also imagine a list as many boxes next to each other, accessible by the name of the list and at the same time by position. These boxes are independent of each other and can contain different values and even, as the list in the picture shows, different types. In the list from the picture, positions 0 and 3 have a `string`, positions 1 and 2 have `int`, etc. We often need to append a value to the end of the list. This is where the *method*[1]



Obr. 1: Example of a 6-element list with indices.

`append` is used. This method adds a new value **at the end** of the list. If we want to access one of the elements of the list, we simply wrap the position in square brackets after the list name, i.e., `<list>[<position>]`. For example, `numbers[0]`.

```
# The program receives a number n, the count of numbers that will follow
# on the next lines. It then reads these numbers and prints them.
n = int(input())
input_list = []
for i in range(n):
  input_list.append(int(input()))

for i in range(n):
  print(input_list[i])

# For example, for the input:
# 3
# 1
# 2
# 3
# The program will output (shortened to one line):
# 1 2 3
```

This program is perhaps the most complex we've encountered in this guide so far. It reads a number n at the beginning, the count of numbers that will follow on separate lines. Then, in a loop from 0 to $n-1$ (inclusive), it reads one number each time using `int(input())` and appends it to the list `input_list` using `append`.

Then the program prints the read numbers. In the `for` loop from 0 to $n-1$, it sequentially accesses the elements of the list at indices 0 to $n-1$ and prints them.

The last loop can be written a bit simpler. The `for` loop itself has a header in the form of `for <variable> in <list>:`. So, the loop variable can take values from any list. This means that the function `range`, which we introduced in the previous chapter, does nothing more than create a list.[2] Let's try it!

```
# We need to use the list function to convert range to a list
# because range is "lazy," and it converts to a list only when it really has to.
print(list(range(5)))
# Output: [0, 1, 2, 3, 4]

print(list(range(0, 10, 2)))
# Output: [0, 2, 4, 6, 8]
```

[1] A method is a function bound to a specific *object*, where an object in Python can be anything (variables, values, functions, etc.). We can call the method using the syntax `<object>.<method>()`. For example, `numbers.append(8)`. You can think of it as if `append` is a regular function that takes the list `numbers` as its first parameter and 8 as its second parameter.

[2] The reality is a bit more complicated since `range` is a *generator*, but the analogy with a list is sufficient.

Now we can print `list` *input_list* from the example where we read n numbers a bit simpler.

```
# We assign values "hardcoded" to the "input_list" to make this code
# executable without an error message.
input_list = [1, 2, 3]

# The variable "x" successively takes values 1, 2, 3.
for x in input_list:
    print(x)
# Output (shortened to one line):
# 1 2 3
```

The above examples assume that the numbers are separated by newlines. What if we get n numbers on a single line? The `input` function would read them all into a single string, and the `int` function would fail because it can only convert **one number from a string**, not multiple.

```
# The program receives a number n, the count of numbers that will follow
# on the next line separated by a space.
n = int(input())
numbers = input()
int(input())
# Error!
# For example, for the input:
# 3
# 3 4 5
# We get an error:
#    ValueError: invalid literal for int() with base 10: '3 4 5'
# This means that the int() function cannot convert '3 4 5' into a single number.
```

Therefore, we use the `split` method to split the numbers in the string by spaces and return a list containing individual numbers (as strings).

```
# The program receives a number n, the count of numbers that will follow
# on the next line, separated by a space.
n = int(input())
# the variable "numbers" will contain the entire input line as a string,
# for example, '3 4 5'
numbers = input()

# "split_numbers" will contain individual numbers in the list as strings,
# because the split method divides the contents of the "numbers" variable by spaces.
# For example: ['3', '4', '5']
split_numbers = numbers.split()

# We need to convert the individual strings in the split_numbers array to ints.
# For example: [3, 4, 5]
int_numbers = []
for x in split_numbers:
  int_numbers.append(int(x))

print(int_numbers)

# Input:
# 3
# 3 4 5
# Output:
# split_numbers: ['3', '4', '5']
# int_numbers: [3, 4, 5]
```

Let's show one more example of using a list, this time containing strings. The program is supposed to print the code of the month from the input.

```
# The program receives a number of the month (indexed from 0) and prints its abbreviation.
months = ["jan", "feb", "mar", "apr", "may", "jun",
          "jul", "aug", "sep", "oct", "nov", "dec"]
n = int(input())
print(months[n])
```

Taskset

Enlarge 2, Sum Low, Count Low

9 Nested Loops

Let's try to create a program that reads a number n and prints the numbers from 0 to $n-1$ on three consecutive lines. This should not be difficult, right?

```
# The program prints numbers from 0 to n - 1 ("n" is the input) on three consecutive lines.
n = int(input())
for i in range(n):
  print(f'␣{i}', end='')
print() # new line
```

```
for i in range(n):
  print(f'␣{i}', end='')
print() # new line

for i in range(n):
  print(f'␣{i}', end='')
print() # new line

# Input:
# 3
# Output:
# 0 1 2
# 0 1 2
# 0 1 2
```

In this program, we introduce a new construct for creating a string: `f'text {<variable>} text {<variable>}'`. By specifying a string with the letter `f`, you enable *interpolation* of variables using curly brackets. The value of the variable appears in the string at the position where its name is in curly brackets. For example, `f'This is {name}. His favorite animal is {animal}.'`.

We presented this example for another reason. You probably already suspect that copying the `for` loop three times is not the best solution. How could we repeat the lines multiple times? Well, by using a loop! In a loop that repeats 3 times, we write a loop that prints the numbers from 0 to $n-1$ each time.

```
# The program prints numbers from 0 to n - 1 ("n" is the input)
# on three consecutive lines using a nested loop.
n = int(input())
for i in range(3):
  for j in range(n):
    # We don't print a space before the first number in the row
    if j > 1:
       print('␣', end='')
    print(i * j, end='')
  print() # new line
```

Two loops are separated from each other in the notation by specifying which is the *outer* loop and which is the *inner* loop. Note that the inner loop must use a different variable than i because otherwise, there would be a collision of names, and the outer loop could be affected by the inner loop. So in the inner loop, we used the name j.

Every time the body of the outer loop begins execution, the inner loop that prints numbers from 0 to $n-1$ is executed again.

9.1 Multiplication Table

Now imagine that we want to create a small multiplication table up to 4. The multiplication table is a simple table that contains the multiplied number of the row and column in any cell.

*	1	2	3	4
1	1	2	3	4
2	2	4	6	8
3	3	6	9	12
4	4	8	12	16

Tabuľka 1: Table of the small multiplication table up to 4.

Now let's write a program that prints the body of such a table. How would we proceed if we wanted to print such a table row by row from left to right? In the row with the number 1, all values of columns (1 to 4) would alternate. Then we would move to the next row with the number 2, and again, all values of columns (1 to 4) would alternate. We would continue like this until the last row.

Notice that for each row, we always go through all the columns. We can imagine this as a loop through the rows from 1 to 4, in which there is an additional *nested* loop through the columns from 1 to 4.

```
# Print a multiplication table of size 5x5.
for i in range(1, 5):
  for j in range(1, 5):
    # Don't print a space before the first number in the row
    if j > 1:
       print('␣', end='')
    print(i * j, end='')
  print() # new line
```

It's worth mentioning the behavior of `break` and `continue` in nested loops. Both commands are related to the *closest enclosing* loop. This means that, for example, in the above example, `break` couldn't escape the *entire* nested loop, but only jump to the outer loop.

```
# The program prints pairs of numbers from 0 to 8 in order from the smallest pairs
# until the first number is divisible by 3 and the second number is also divisible by 2.
for i in range(8):
  for j in range(8):
    # Error! We can't jump out of the outer loop!
    if i % 3 == 0 and j % 2 == 0:
      break
    # Prints i and j separated by a space, equivalent to f'{i} {j}'.
    print(i, j)

# Output:
# 0 0
# 0 1
# ...
# 0 7
# 1 0
# ...
# 3 1
# ! 3 2 won't be printed, but the program continues (error!)
# 4 0
# 4 1
# ...
```

If we used `continue` instead of `break` in the above example, the next printed pair after skipping (3 2) would be (3 3) because the program would continue (error!) without printing.

10 List of Lists

Now that we know how to nested loops, it probably won't surprise you that lists can also be nested :) For example, here's a list of lists containing information about a few animals.

```
animals = [
  ["Bax", "dog", "black␣and␣white"],
  ["Queen", "cat", "white"],
  ["Fero", "tomcat", "brown␣and␣white"]
]
```

Note that the list `animals` contains 3 more lists. Each of these 3 lists describes one animal. The list for a specific animal contains its name, type, and color.

We index nested lists the same way as the one-dimensional ones. So when we index into the `animals` list, we get another list.

```
# animals = ...from the previous example...

print(animals[1])
# Output (list): ["Queen", "cat", "white"]
```

This list can be indexed again.

```
# animals = ...from the previous example...

# Color of the first (zeroth) animal
print(animals[0][2])
# Output: "black and white"
```

Now let's try to write a program that prints the names of all animals.

```
# animals = ...from the previous example...

for animal in animals:
  print(animal[0])

# Output (condensed to one line):
# Bax Queen Fero
```

In the above for loop, we iterate over all `animals`. The variable `animal` will, therefore, over time, contain lists `[„Bax“, „dog“, „black and white“]` through `[„Fero“, „tomcat“, „brown and white“]`. From these lists, we then print the 0-th value, which is the name.

How to read a *two-dimensional list*[3] (a table of numbers)? Using nested loops!

[3] A two-dimensional list may sound intimidating, but with what we already know, the explanation is easy. It's simply a list of lists. The term „two-dimensional“ is used because it's essentially a table where we can move both in rows and columns.

```
# The program receives the numbers m and n on one line, the number of rows, and the number of columns of the table.
# Then it receives the individual values of the table.

dimensions = input().split()
# Shortcut for
# m = int(dimensions[0])
# n = int(dimensions[1])
m, n = int(dimensions[0]), int(dimensions[1])

table = []
for i in range(m):
  row_str = input().split()
  row_int = []
  for x in row_str:
    row_int.append(int(x))
  table.append(row_int)

print(table)

# Input:
# 3 4
# 1 2 3 4
# 5 6 7 8
# 9 10 11 12
# Output:
# [
#  [1, 2, 3, 4],
#  [5, 6, 7, 8],
#  [9, 10, 11, 12]
# ]
```

10.1 Examples

Difference, Sum Pyramid, Exchange, Rotation

11 Functions

Imagine that in your program, you need to do the same thing frequently. For example, you need to check if a number is even and print the answer in multiple places. Of course, you can use `if` and `print` every time you need this, resulting in something like this:

```
if x % 2 == 0:
    print("Even number.")
else:
    print("Odd number.")
#
#
#
if a % 2 == 0:
    print("Even number.")
else:
    print("Odd number.")
#
#
# and so on
```

While this achieves what we want, it leads to longer, less readable code with more places to make mistakes. Don't despair; functions will save us. The above program can be equivalently written as follows:

```
def printParity(number):
    if number % 2 == 0:
        print("Even number.")
    else:
        print("Odd number.")
#
#
printParity(x)
#
#
printParity(a)
#
# and so on
```

If you compare the two examples above, it should be intuitively clear what the functions do. We have already encountered functions – `print` is a prime example of a function. Now, what is a function? It's multiple statements linked into one unit, forming a kind of super-statement that does something useful. Remember, a function can have several dozen lines if it does something complex. The first thing we can see in the example above is that the function has parameters. *Parameters* are placed in parentheses after the function and are a way for the function to receive some information. In the above example, the function `printParity` has one parameter named `number`. This parameter is there because we want to pass a number to the function, which is

needed to decide what to print. When we want to use the function, we say we're *calling* it, and in the program, we see two calls to the `printParity` function. When calling a function, we pass our number as a parameter.

In addition to taking parameters, functions can also return something. The function above doesn't return anything. If we wanted to see an example of a function returning something, it might look like this:

```
def add(a, b):
    return a + b
# usage:
x, y = 10, 20
result1 = add(x, y)
result2 = add(x, 77)
result3 = add(5, 11)
```

Above, we see a very simple function that just adds its parameters and returns them using the `return` keyword.

One important thing is how variables behave after being passed as parameters. If we write a simple program like this:

```
def change(x):
    x = 30
a = 10
print(a)
change(a)
print(a)
```

After running it, we find that the value of the variable a did not change after the change in the `change` function. This is because only a copy of the variable a was passed to the function, and this copy is discarded after the function execution. However, things get a bit more complicated when we run the following program.

```
def change(x):
    x[0] = 30
a = [1, 2, 3]
print(a)
change(a)
print(a)
```

We might expect that a won't change this time. However, in this case, a is a `list`. Why did the value of a change this time? It's because, in this case, the actual list a was passed to the function, not a copy of it. It's a bit illogical why Python behaves differently for different types. For example, lists are generally quite large, taking up more space, and therefore copying them is slow, so Python tries to avoid it. It's essential to be careful with this behavior and, when necessary, test how Python behaves for a specific type.

Taskset

Buttons

12 Hooray!

If you made it this far and completed all the tasks, you have a solid foundation in the Python language. We believe that your programming journey doesn't end here. You know, it's never goodbye, it's...

See you later :)