

Welcome to Python Academy

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [WELCOME TO PYTHON ACADEMY](#)

Hi there! We are excited to welcome you in our Python Academy. This course consists of 12 in-person lectures and self-study in Online portal, where you can find dozens of exercises, articles about theory and complex projects. In case of any problem, feel free to use our live chat (blue icon in the bottom right corner).

Let's start with lesson 1. We will discuss the basics of Python:

- what is it,
- how to install it,
- how to write scripts in Python,
- Python data types
- and more.

48% z Lekce 1



Course Structure

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [COURSE STRUCTURE](#)

Python Academy consists of:

12 in-person lessons

We meet once a week, always at 18:00. Our focus during in-person lessons is to explain new concepts and answer your questions - **OnSite** part. Because we are limited just to 3 hours a week, there's a need to continue learning at home.

So in order to be successful, you may need to revise what we've discussed in the **Theory** sections, and then you'll certainly need to test your knowledge in the **Quizzes** and try new concepts on tasks inside **Home exercise** sections.

Our data show that without exceptions, best graduates are the ones who completed most exercises. If you don't have time during one week, that's fine. Try to work on exercises another week. Just think about the fact, that practice is the king.

Additionally, sometimes there will be **Theory +** sections. If you aim to be serious about programming, it is a good idea to take a look at this section too ;). Here's an [example](#) of such section.

And lastly, there will be few exercises which will have the **[H]** sign which stands for **Hacker**. These exercises are still easy for some, but harder for others. They are not necessarily super advanced, but maybe just a bit larger in scope. If you feel up for a challenge, we recommend those as well. Here's an [example](#) of such lesson - notice the **[H]** in the title.

To conclude, each lesson consists of the:

- Introduction

48% z Lekce 1

- Theory,
- Theory +,
- Quiz,
- and Home Exercises.

3 big projects

At the end of each month you'll get to work on a project that should test the knowledge you have acquired so far. We'll be there to give you feedback, shall you need it :)

Every project lesson includes a review test, which you should complete before diving into the project itself ;)

- [Project 1 review test](#)
- [Project 2 review test](#)
- [Project 3 review test](#)

2 small projects

You'll also have 2 optional projects at your disposal - the Black Jack game and the HR System. These projects are additional practise to test your programming capabilities.

The Black Jack game includes the - [last review test!](#)

Basics of command line

PYTHON ACADEMY / 1. INTRO TO PROGRAMMING / BASICS OF COMMAND LINE

We will go through the very basics of command line here. If you're familiar with it, feel free to skip this part :). We will distinguish between 2 types of command line:

In **Windows** we simply call it command line - **cmd**:

48% z Lekce 1

In **Unix** (Linux, Mac) we'll call it **terminal**:

```
$
```

We **don't** write the symbols `>` and `$` into our command line. They are already there by default.

Where do I find command line?

- Windows : **Start > write „cmd“ > Command Prompt**
- Linux : **Super > search Terminal**
- Mac : **Applications > Utilities > Terminal**

Current directory

After running the command line, we can write our first command, which will let us know the directory where we are at the moment.

Windows:

```
> cd  
C:\Users\pepa
```

Unix:

```
$ pwd  
/home/pepa/
```

Directory content

Next we can print out what is located in the directory.

Windows:

```
> dir  
Directory of pepa  
29.10.2018  09:29      <DIR>          Desktop
```

48% z Lekce 1

11.10.2018 11:57 <DIR> FAVORITES

Unix:

```
$ ls
Applications
Desktop
Downloads
Music
```

Changing current directory

- **cd** - without any additional info, this command will print out the current working directory
- **cd ** - this command will get you back to you root directory
- **cd Desktop** - this will get you to Desktop
- **cd..** - this will get you one directory back

Windows:

```
> cd Desktop
> cd
C:\Users\pepa\Desktop
> cd \
> cd
C:\
> cd Users\pepa\Desktop
> cd
C:\Users\pepa\Desktop
```

Unix:

```
$ cd Desktop
$ pwd
/home/pepa/Desktop
$ cd /
$ pwd
```

48% z Lekce 1

```
$ pwd  
/home/pepa/Desktop
```

There's much more commands, but these will do the trick for now.

Installing Python

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [INSTALLING PYTHON](#)

Now let's look how to install Python to your computer. We'll go over 3 different OS (Operating Systems) starting with Windows.

Windows

You can download Python installer from the [Python website](#). Choose the most up-to-date version of Python 3. It is usually the link at the top of the list.

Looking for a specific release?

Python releases by version number:

Release version	Release date	Click for more	
Python 3.7.1	2018-10-20	Download	Release Notes
Python 3.6.7	2018-10-20	Download	Release Notes
Python 3.5.6	2018-08-02	Download	Release Notes
Python 3.4.9	2018-08-02	Download	Release Notes
Python 3.7.0	2018-06-27	Download	Release Notes
Python 3.6.6	2018-06-27	Download	Release Notes
Python 2.7.15	2018-05-01	Download	Release Notes
Python 3.6.5	2018-03-28	Download	Release Notes

[View older releases](#)

When you click on the latest release you get to a page where at the bottom, you will find multiple Python installation packages to download. You should select only one - based on Windows system type - 64 or 32 bit. So let's say you have the 64 bit:

48% z Lekce 1

Gzipped source tarball	Source release		99f78ecbfc766ea449c4d9e7eda19e83	22802018	SIG
XZ compressed source tarball	Source release		0a57e9022c07fad3dad2eef58568edb	16960060	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	ac6630338b53b9e5b9dbb1bc2390a21e	34360623	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	b69d52f22e73e1fe37322337eb199a53	27725111	SIG
Windows help file	Windows		b5ca69aa44aa46cdb8cf2b527d699740	8534435	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	74f919be8add2749e73d2d91eb6d1da5	6879900	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	4c9fd65b437ad393532e57f15ce832bc	26260496	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	6d866305db7e3d523ae0eb252ebd9407	1333960	SIG
Windows x86 embeddable zip file	Windows		aa4188ea480a64a3ea87e72e09f4c097	6377805	SIG
Windows x86 executable installer	Windows		da24541f28e4cc133c53f0638459993c	25537464	SIG
Windows x86 web-based installer	Windows		20b163041935862876433708819c97db	1297224	SIG

To find out which system type you have, you will need to go to: **Start -> Control Panel -> System and Security -> System -> System type**.

You will most probably want to choose the 64-bit version.

The screenshot shows the Windows 'System' settings window. The navigation pane on the left includes 'Control Panel Home', 'Device Manager', 'Remote settings', 'System protection', and 'Advanced system settings'. The main content area is titled 'View basic information about your computer' and displays the following details:

- Windows edition:** Windows 10 Home, © 2018 Microsoft Corporation. All rights reserved.
- System:**
 - Processor: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz 2.20 GHz
 - Installed memory (RAM): 8.00 GB
 - System type: 64-bit Operating System, x64-based processor** (highlighted with a red box)
 - Pen and Touch: No Pen or Touch Input is available for this Display
- Computer name, domain and workgroup settings:**
 - Computer name: DESKTOP-R6TCNBD
 - Full computer name: DESKTOP-R6TCNBD
 - Computer description:
 - Workgroup: WORKGROUP
- Windows activation:** Windows is activated. Read the Microsoft Software Licence Terms. Product ID: 00326-10000-00000-AA087.

At the bottom, there are links for 'See also' and 'Security and Maintenance'.

While installing, you have the possibility to tick the following options. **Please tick them.**

48% z Lekce 1

Afterwards you can click on **Install Now**.



Mac

Install Homebrew Package Manager on your Mac. Open the terminal window and paste following command in:

```
/usr/bin/ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Then install python using the command below:

```
brew install python3
```

Linux

If you have **Ubuntu** you can write into your terminal:

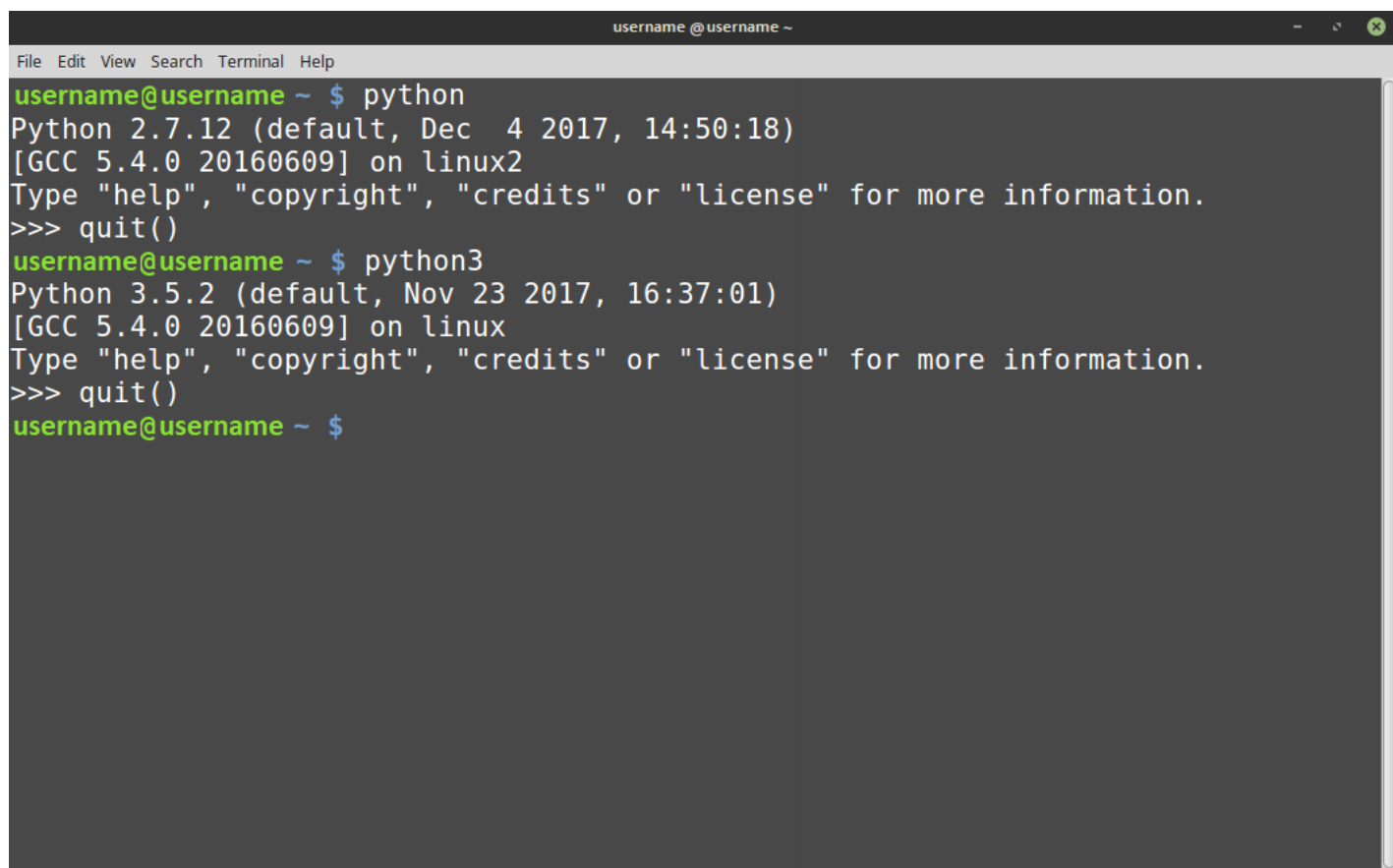
48% z Lekce 1

In case of **CentOS** or **Fedora** you don't need to do anything. Python 3 is by default installed on Fedora and latest CentOS.

Running Python in Command Line

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [RUNNING PYTHON IN COMMAND LINE](#)

All you need to do now to run Python is to open your terminal and enter the command **python** or **python3**.



```
username @username ~  
File Edit View Search Terminal Help  
username@username ~ $ python  
Python 2.7.12 (default, Dec 4 2017, 14:50:18)  
[GCC 5.4.0 20160609] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> quit()  
username@username ~ $ python3  
Python 3.5.2 (default, Nov 23 2017, 16:37:01)  
[GCC 5.4.0 20160609] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> quit()  
username@username ~ $
```

Frequent problem

The following text concerns you, if you have Windows and you were not able to run Python after entering one of the commands above.

48% z Lekce 1

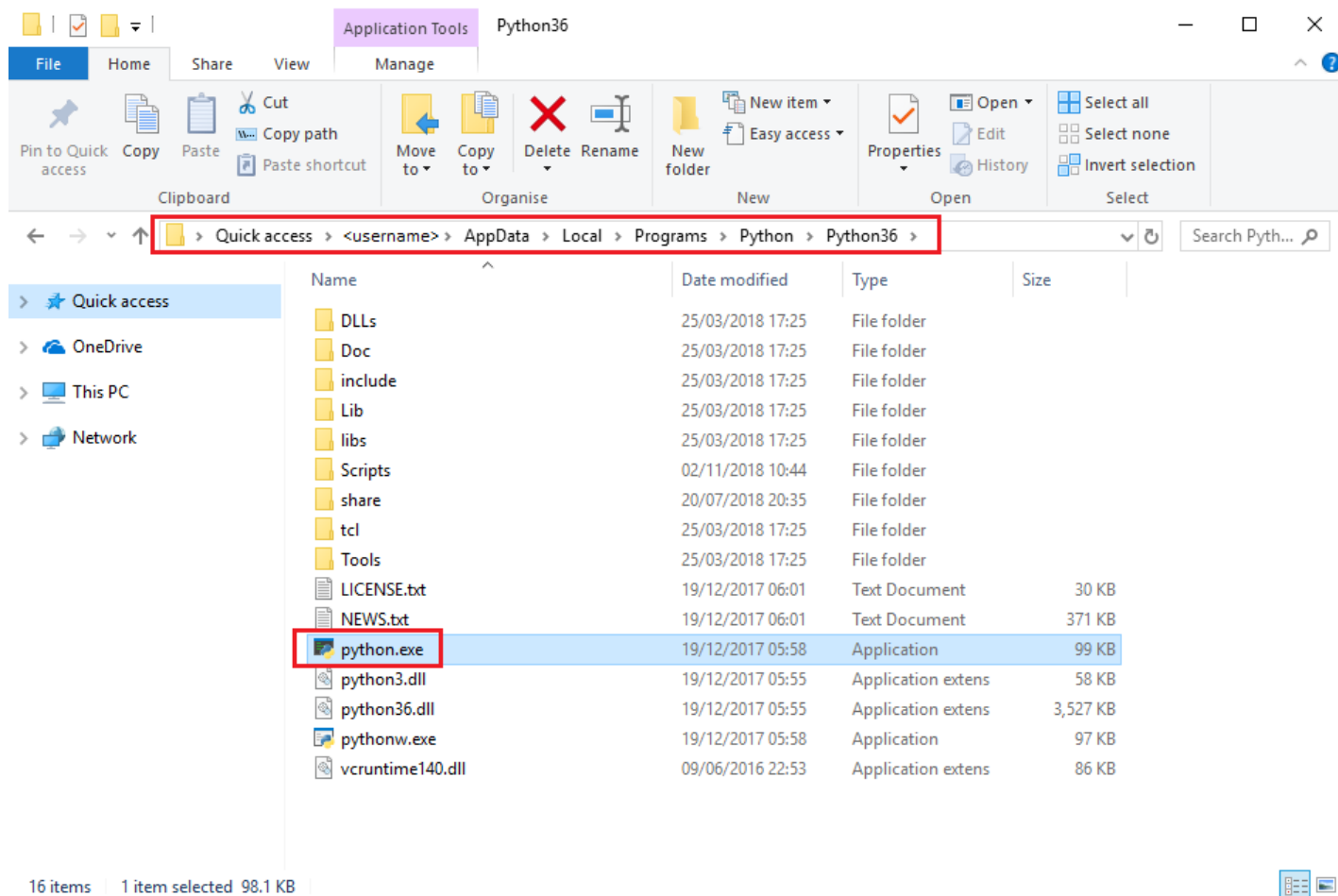
cases it applies to windows. Therefore we will be addressing here only windows.

Ideally, this problem should not occur, because we have set this variable in the installation earlier. However, let's go through the steps, how to set the variable manually, just in case.

Solution to the problem

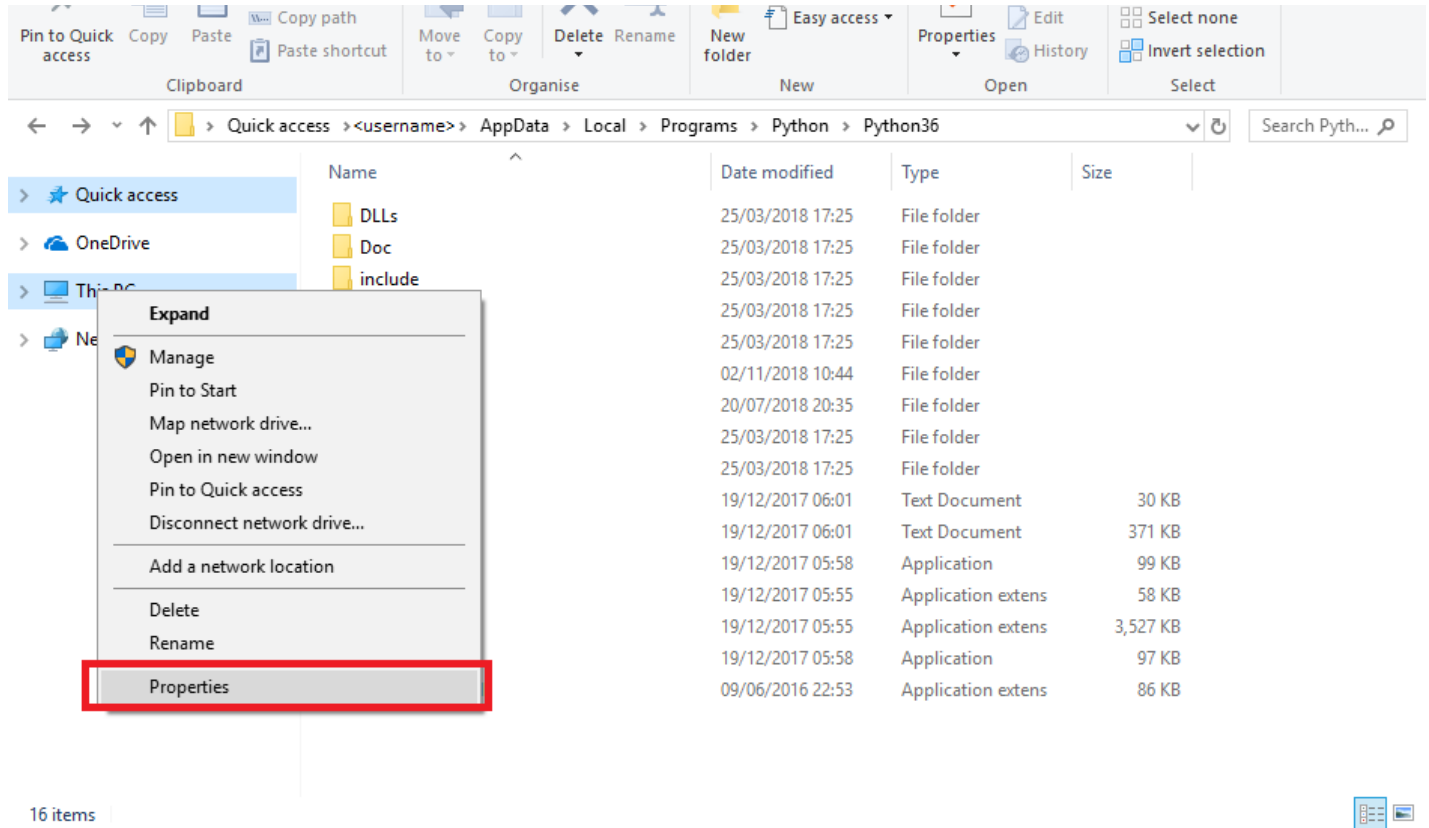
Find out where did you install Python in you computer. Usually it is located at **C:\Users\<username>\AppData\Local\Programs\Python\Python36**

Copy the path to the clipboard (ctrl + C).



Right-click on **This PC** and then **Properties**.

48% z Lekce 1



You'll get to **Control Panel\System and Security\System** where you click on **Advanced system settings**.

48% z Lekce 1

Control Panel Home


- Device Manager
- Remote settings
- System protection
- Advanced system settings**

View basic information about your computer

Windows edition

Windows 10 Home

© 2018 Microsoft Corporation. All rights reserved.



System

Processor:

Installed memory (RAM):

System type:

Pen and Touch: No Pen or Touch Input is available for this Display

Computer name, domain and workgroup settings

Computer name: [Change settings](#)

Full computer name:

Computer description:

Workgroup:

Windows activation

Windows is activated [Read the Microsoft Software Licence Terms](#)

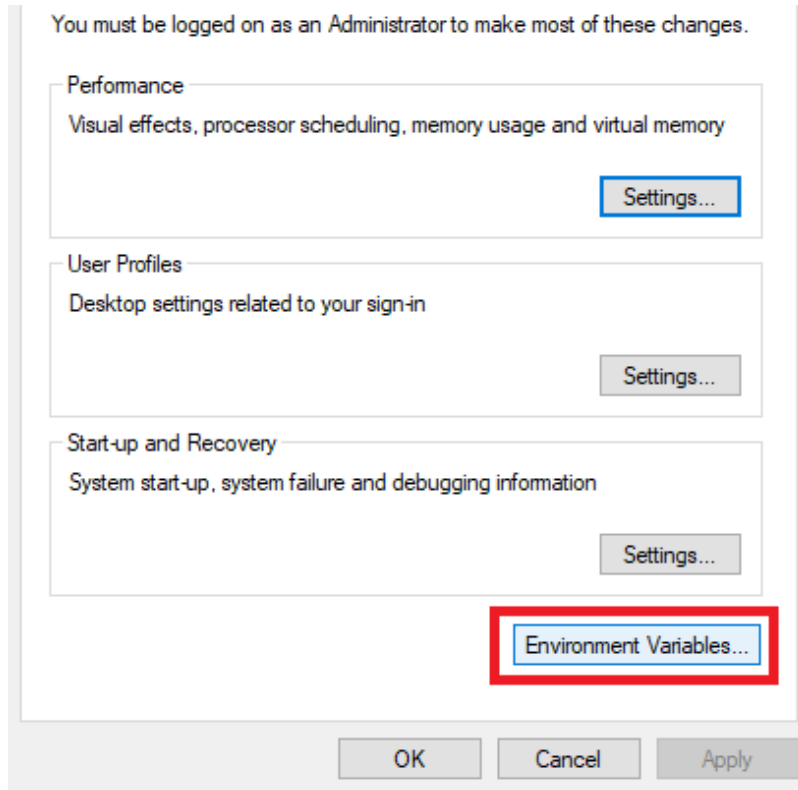
Product ID: [Change product key](#)

See also

[Security and Maintenance](#)

This gets you to **System Properties**. In the tab **Advanced** click on **Environment Variables...**

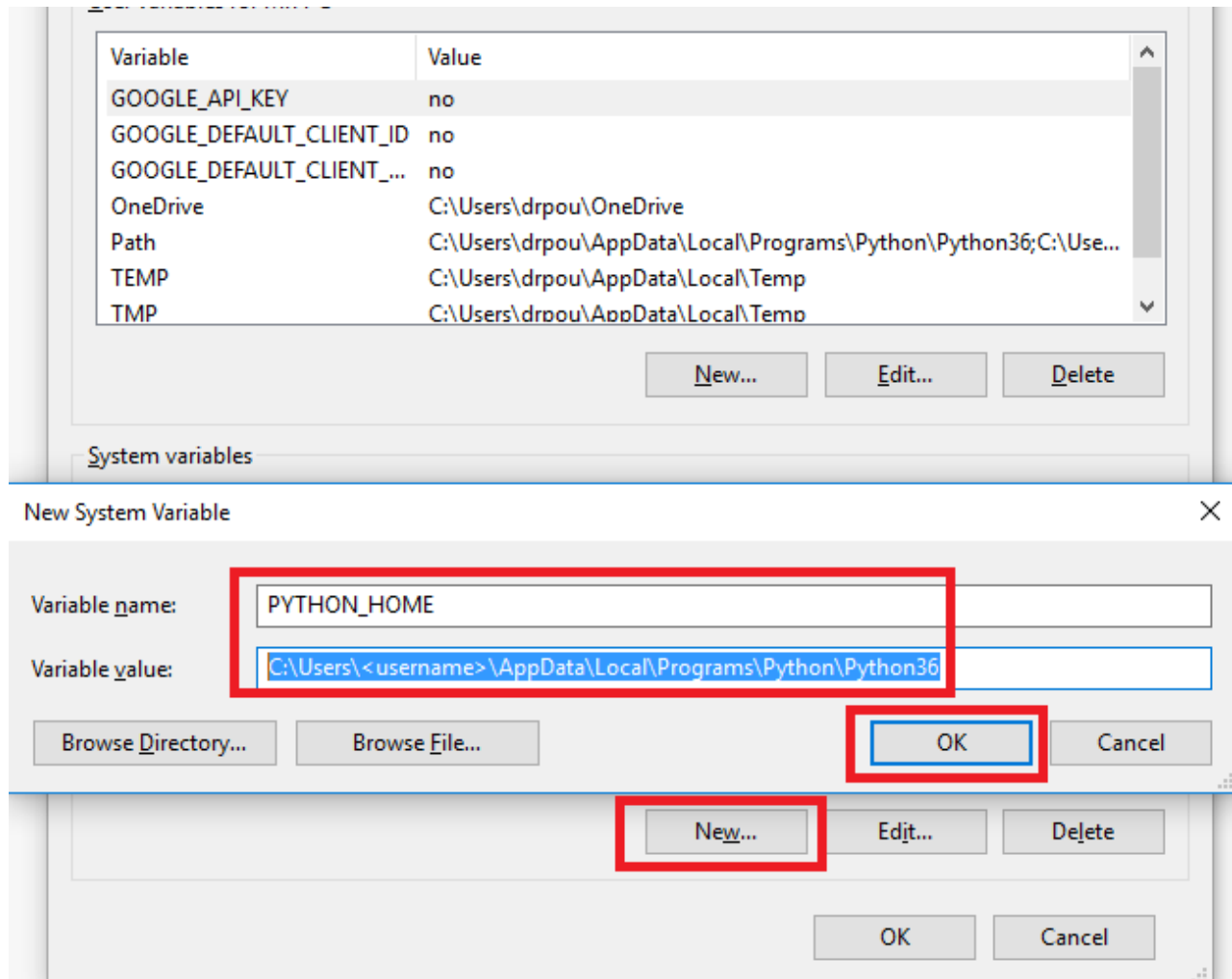
48% z Lekce 1



In the window **Environment Variables** click in the section **System variables** on **New...**

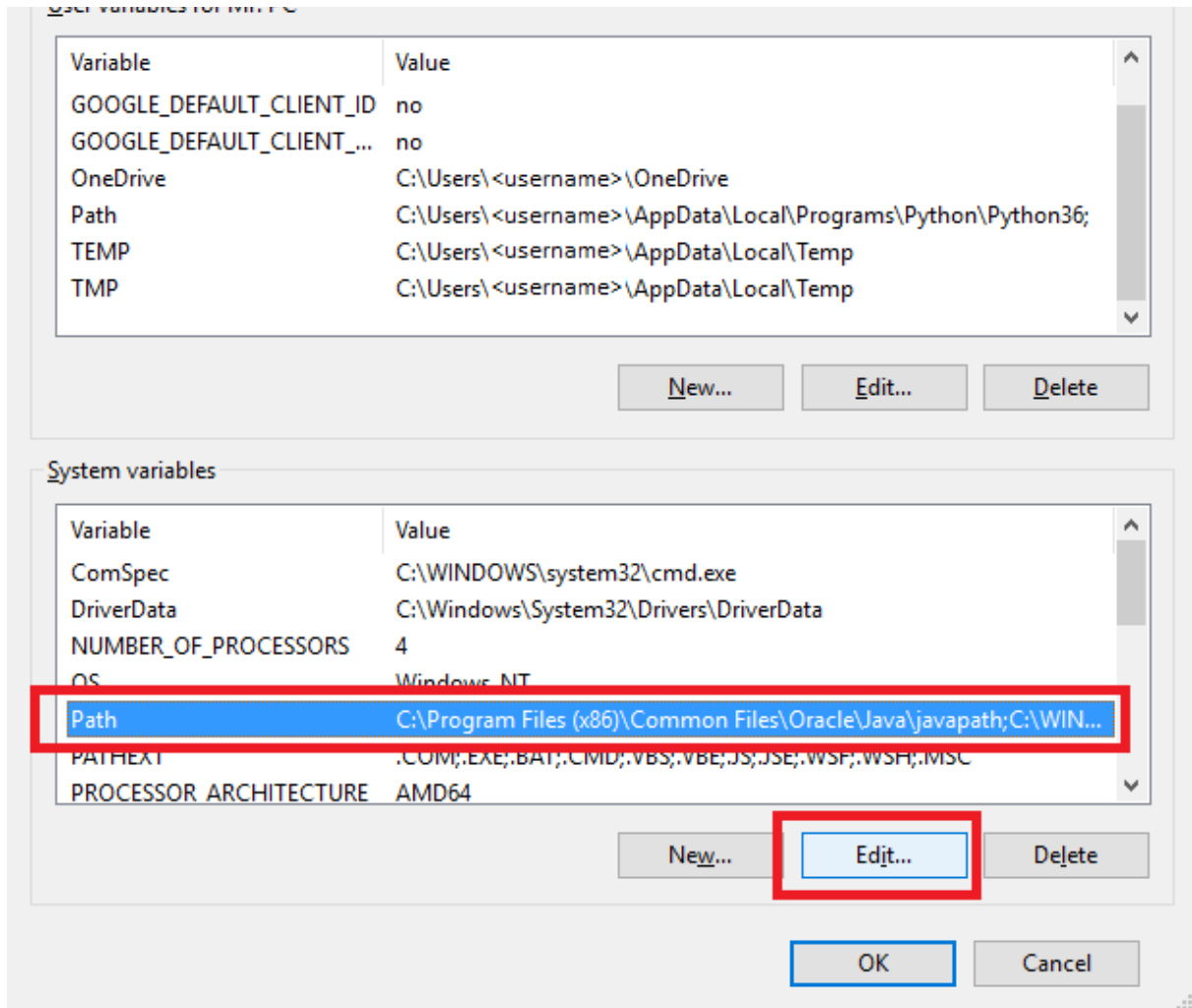
Enter a name of a variable, f.e. **PYTHON_HOME** and as a value insert (ctrl+V) the path to you Python that you've copied earlier.

48% z Lekce 1



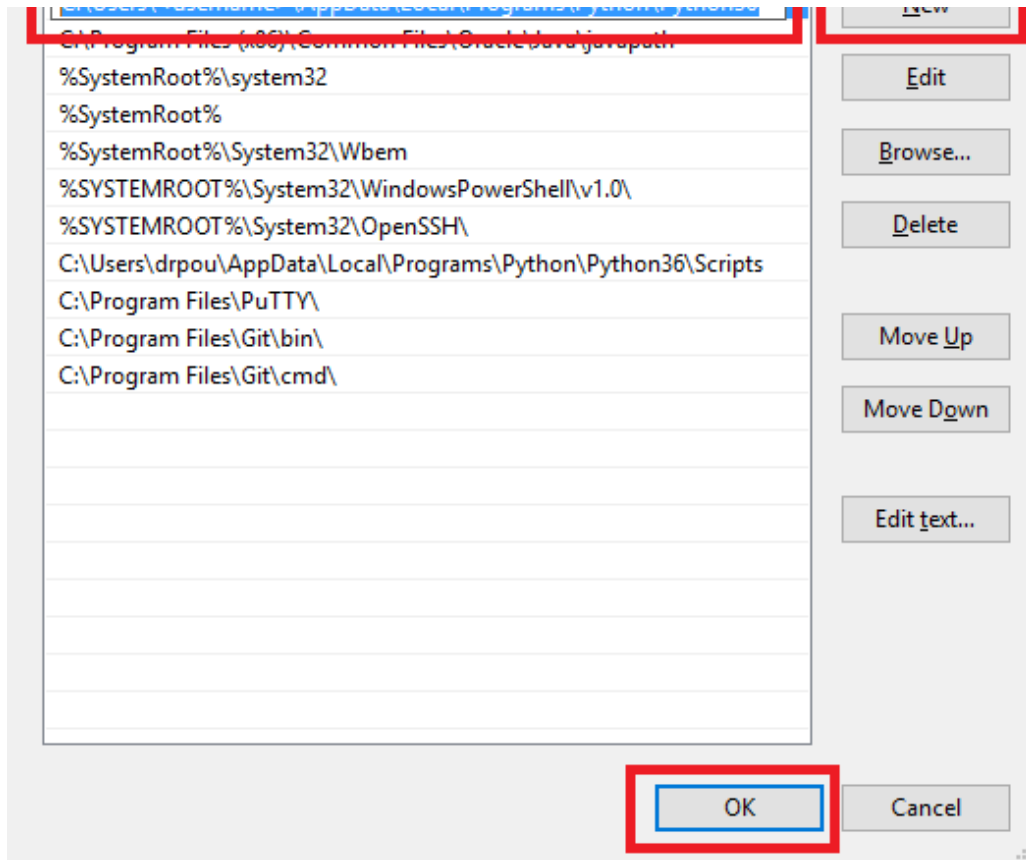
Now we find the variable **Path** in the **System variables** section and click on **Edit**.

48% z Lekce 1



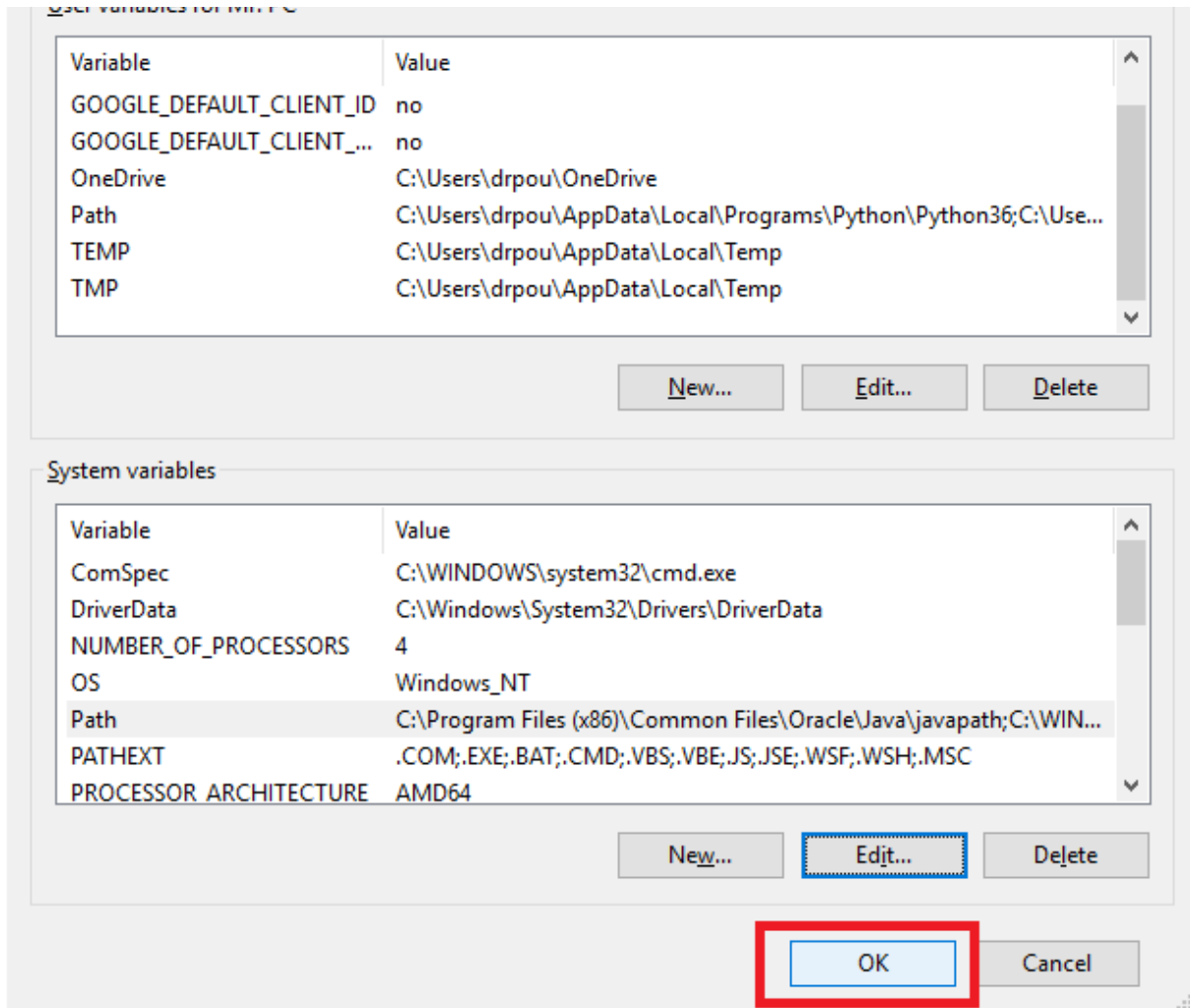
Click on **New** and insert the path to Python again and then press **OK**.

48% z Lekce 1



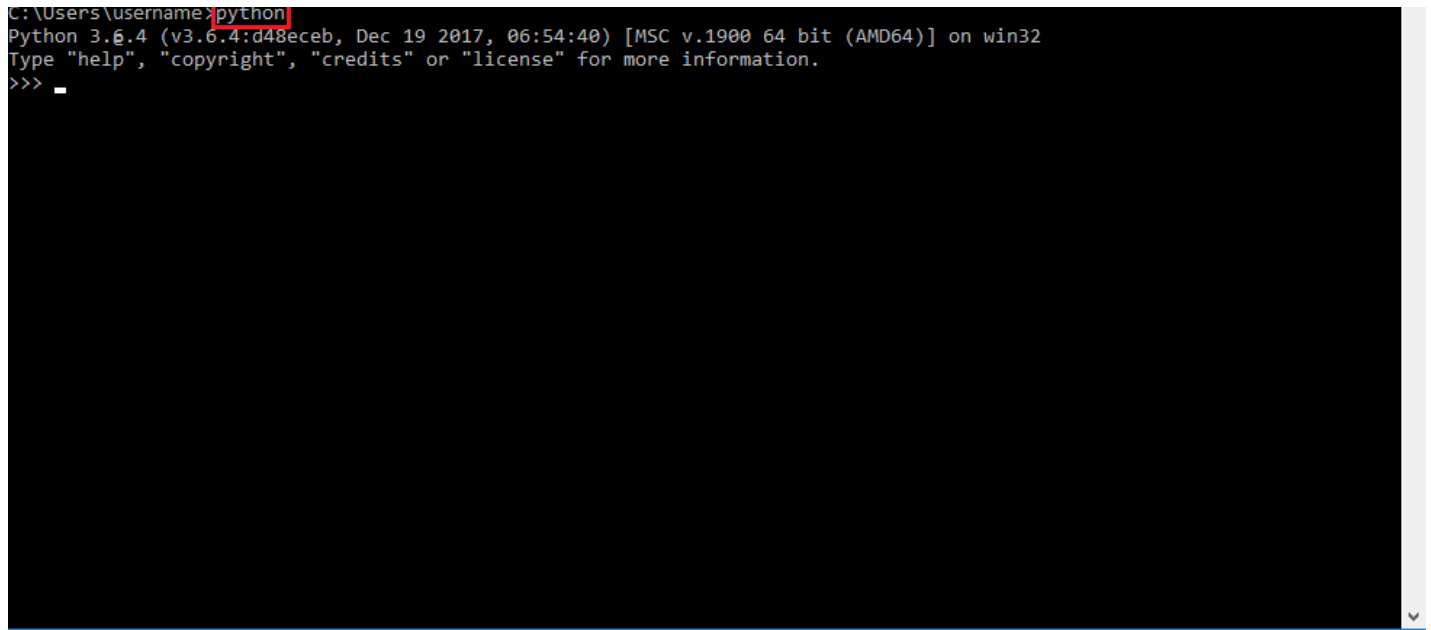
Then again in the windows **Environment Variables** we press **OK**.

48% z Lekce 1



Just to be sure that everything works, we can open our command line and write the command **python**

48% z Lekce 1



```
C:\Users\username>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Installing editor

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [INSTALLING EDITOR](#)

Editor is simply a program for text editing - in our case for code editing. As editor is one of the basic programming tools, let's see how we can install it.

We recommend the following editors:

- [Atom](#)
- [Sublime Text 3](#)
- [Pycharm](#)

We do not recommend PyCharm for beginners.

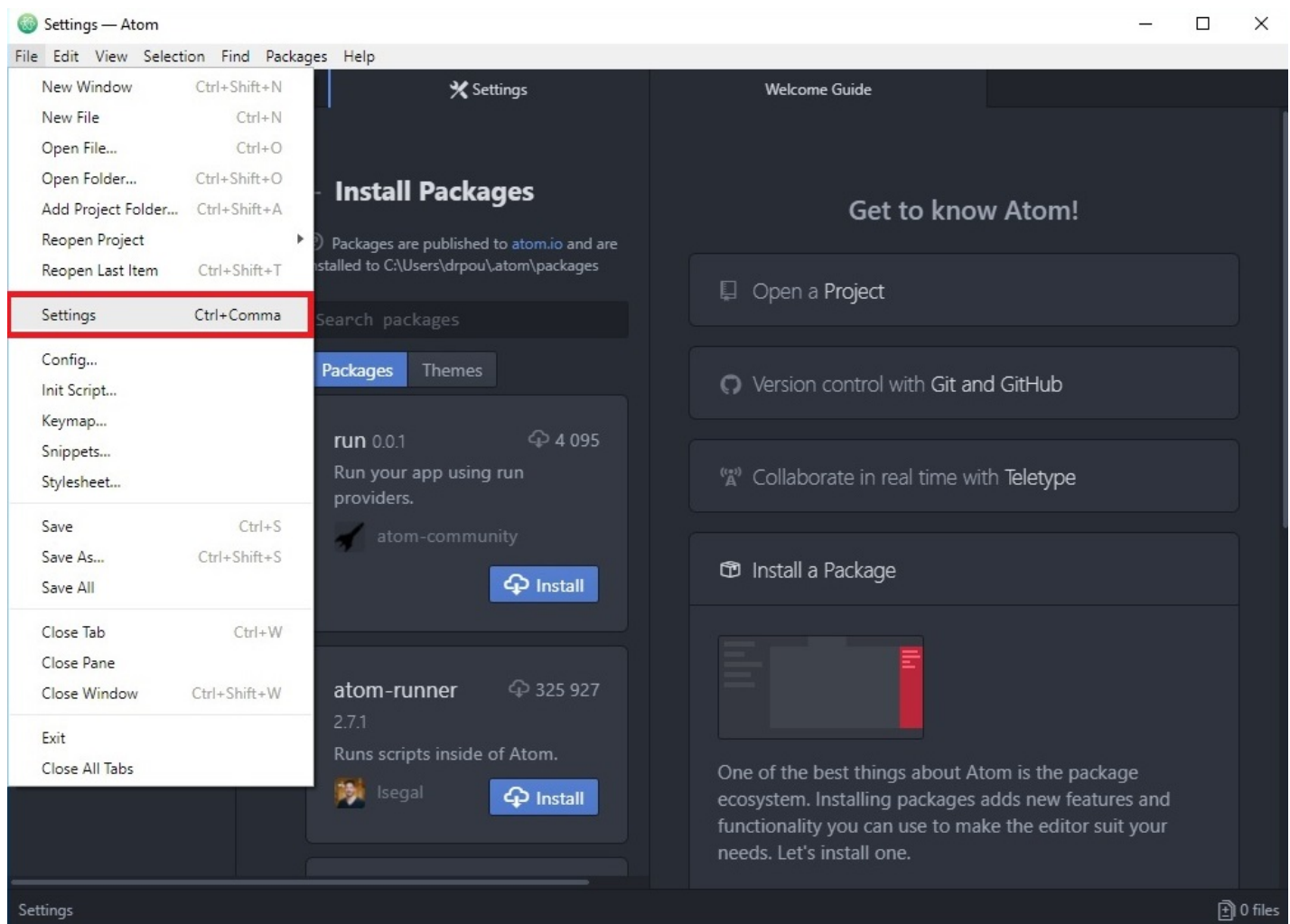
In case of Atom and Sublime Text 3, all you need to do is to download it, click through the installation and run it.

48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [RUNNING PYTHON IN EDITOR](#)

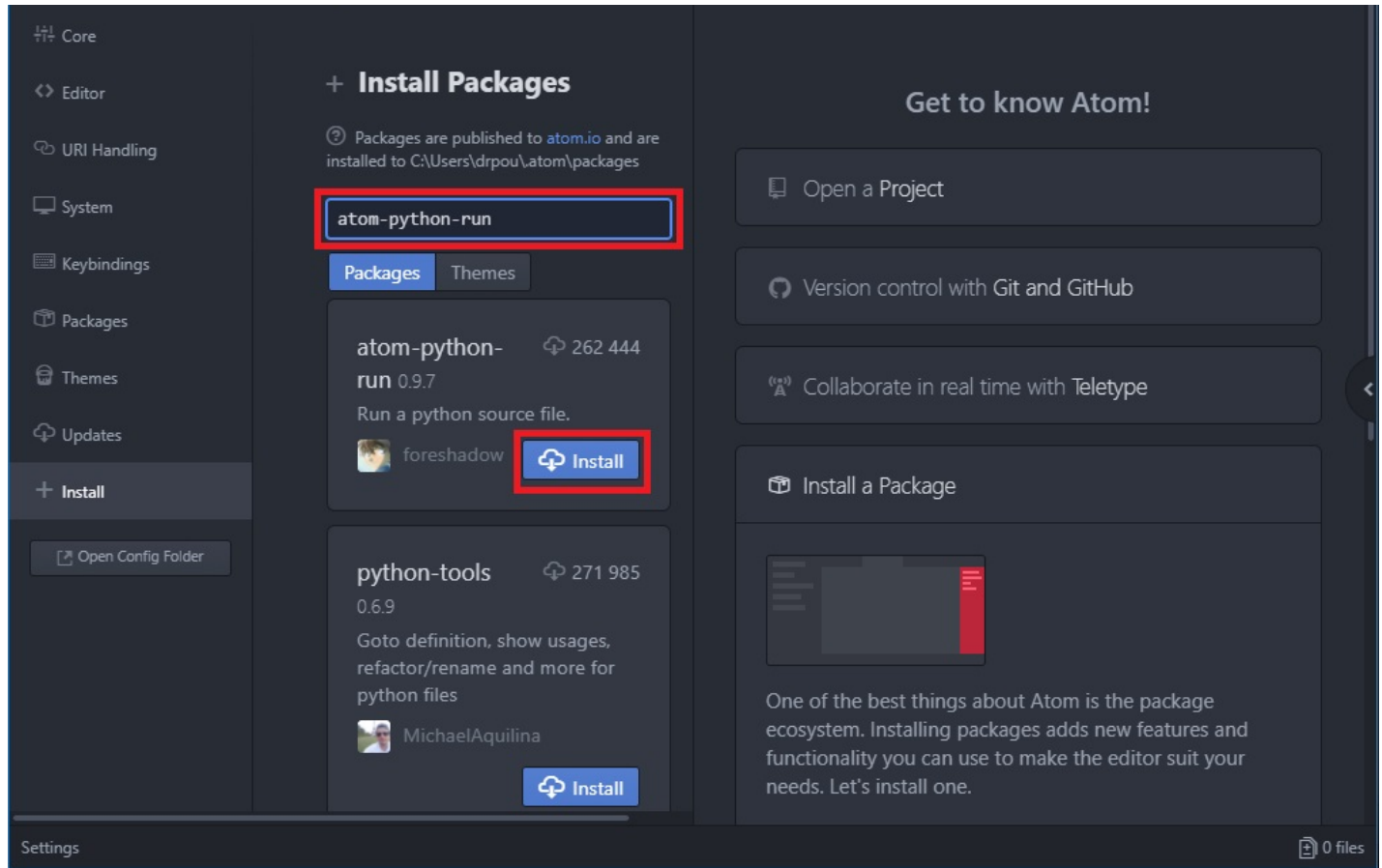
In Sublime Text as well as in PyCharm, it is possible to run the Python code directly in the editor using the keyboard shortcuts, or the buttons of the editor. However, in **Atom** we have to install an extra package. Let's look at how to do that:

We run Atom and in the tab **File** we click on **Settings**.



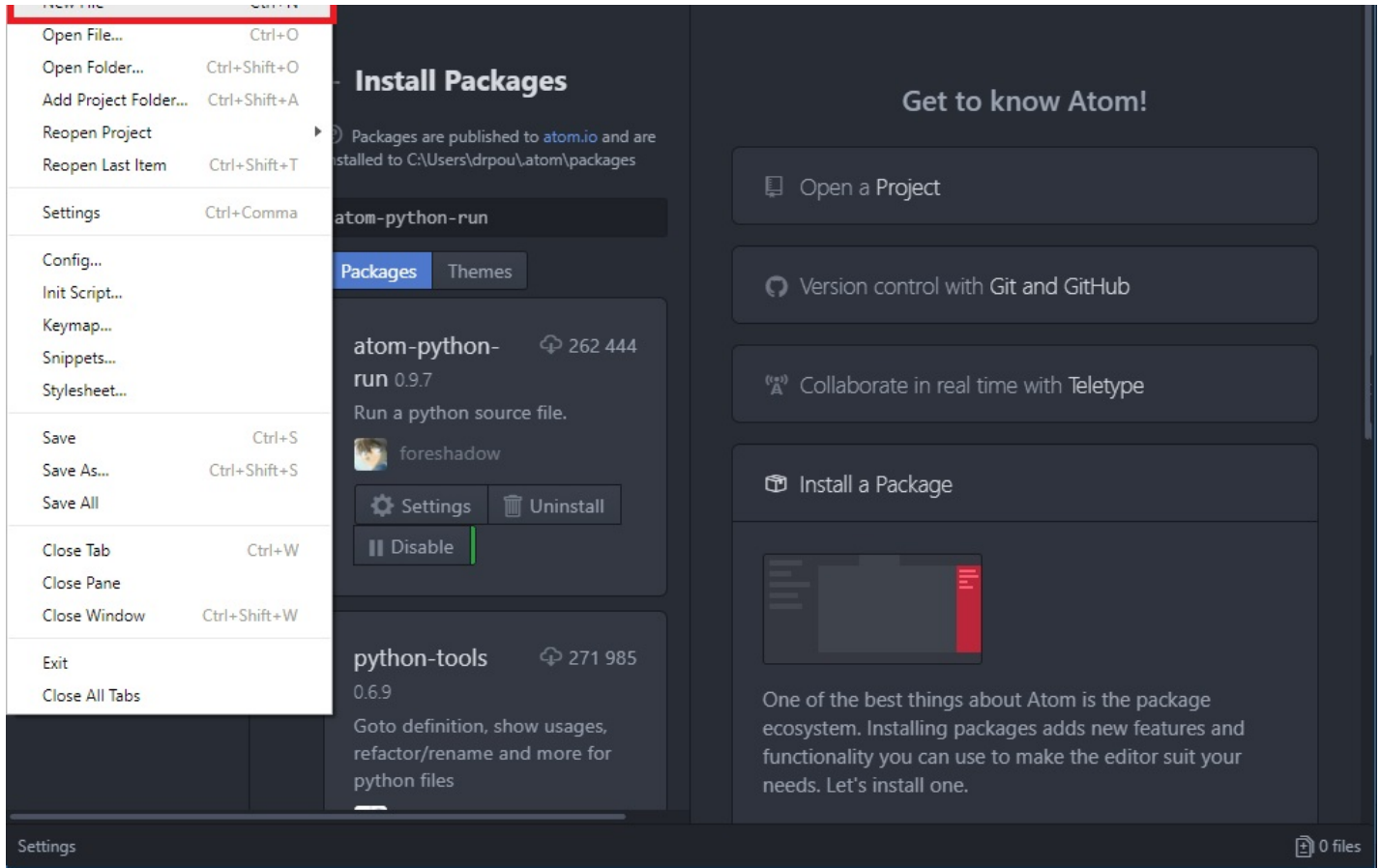
Then it is the option **Install** > **atom-python-run** (search) > **Install**.

48% z Lekce 1



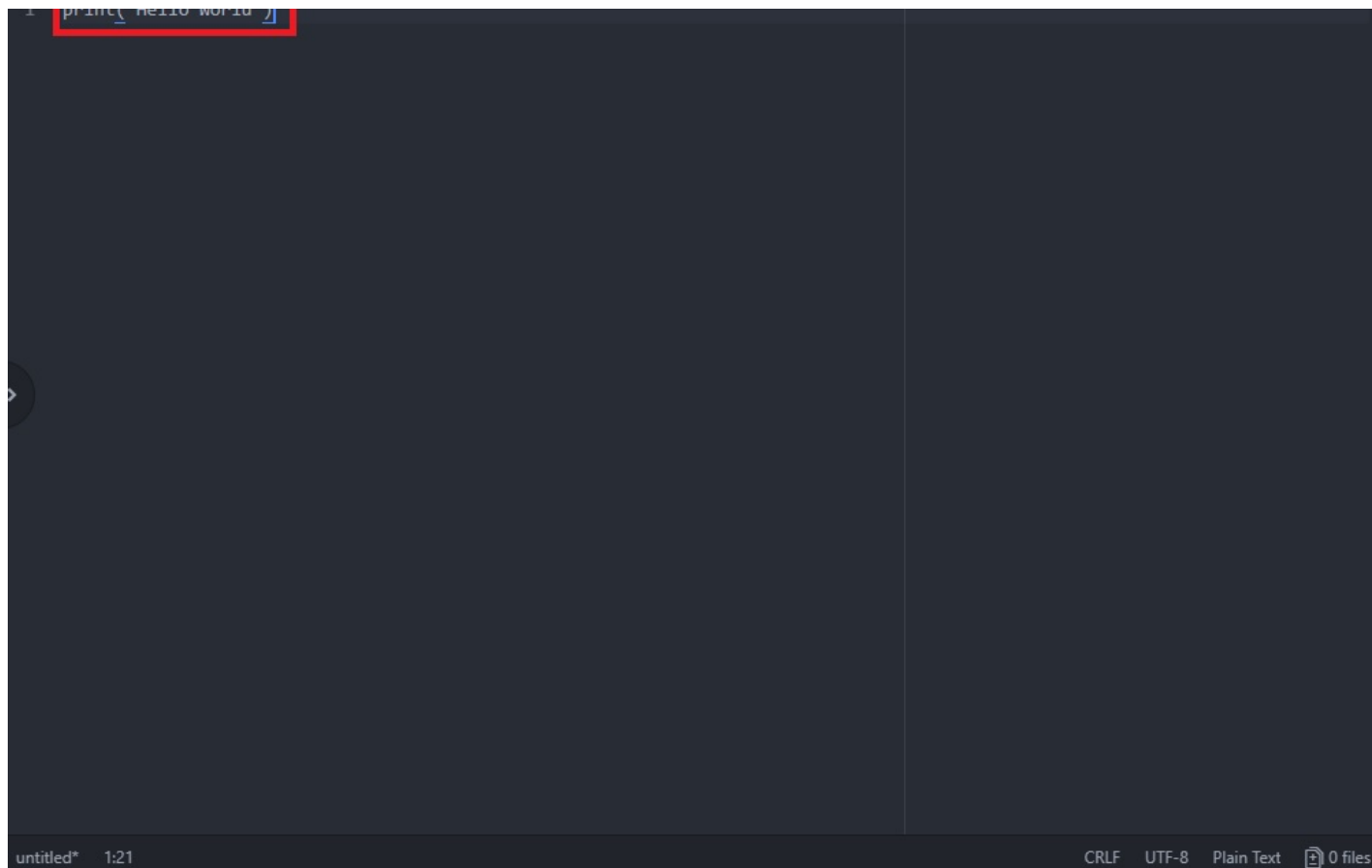
After installation we click on the tab **File > New File**. Surprisingly, this opens a new file which will have some default name - untitled f.e.

48% z Lekce 1



We write the following text in the file - `print('Hello World')`

48% z Lekce 1

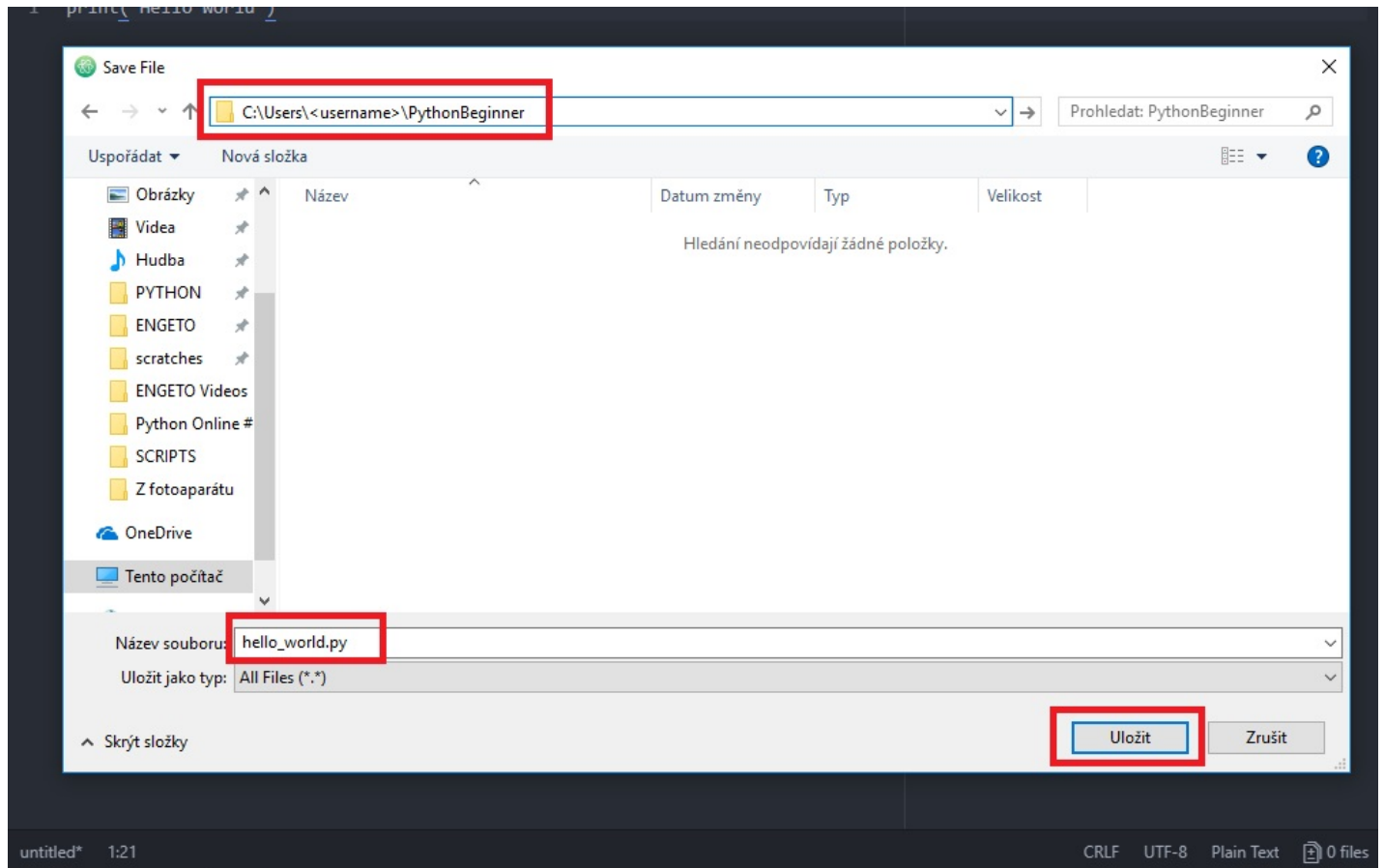


Then we go to **File > Save as...** and save the file to the directory **PythonBeginner**, which we need to create first, of course. Create the directory wherever you like. We recommend the Home Directory:

Operating System	Home Directory
Windows	C:\Users\<username>\PythonBeginner
Linux	/home/<username>/PythonBeginner
Mac	/Users/<username>/PythonBeginner

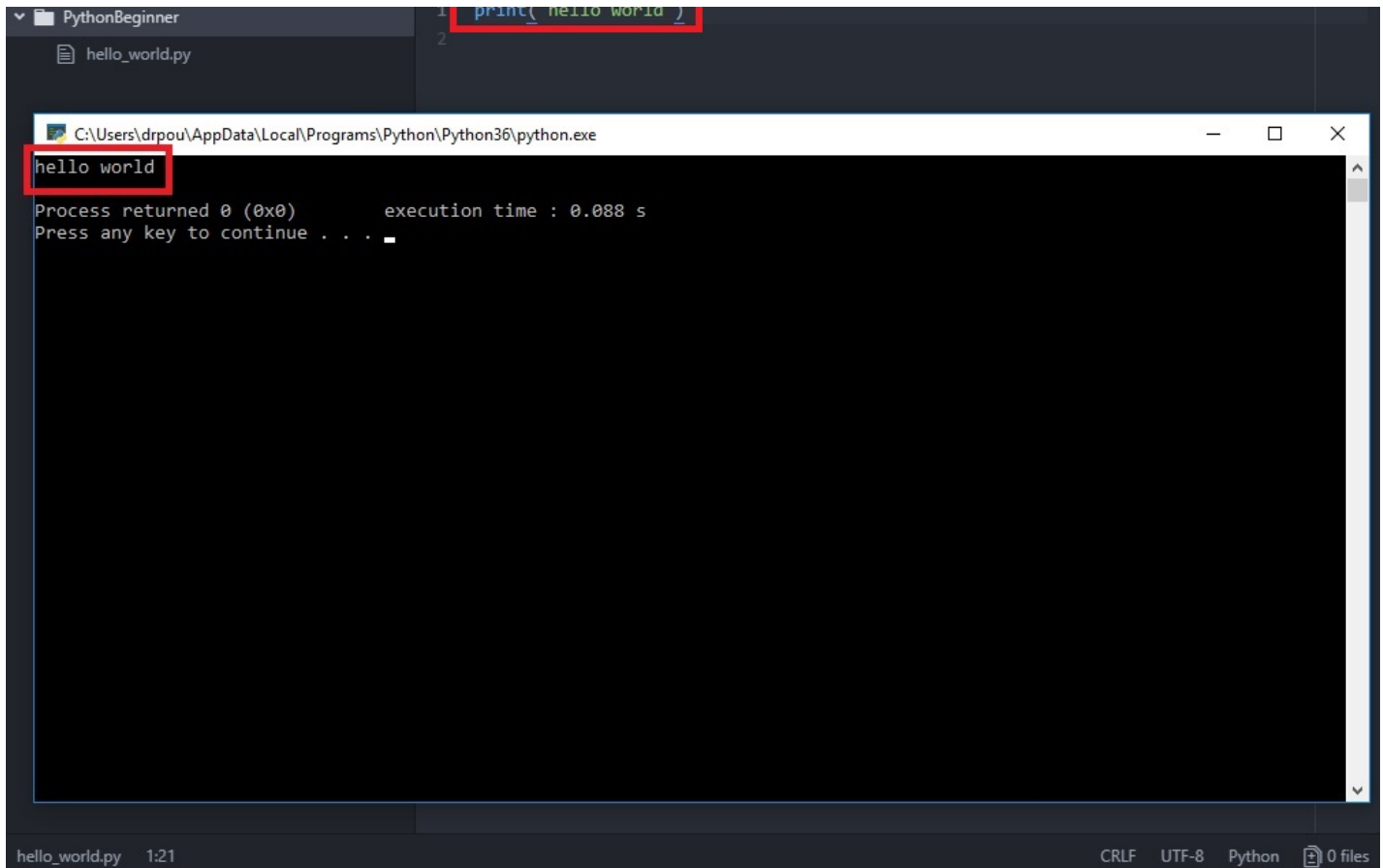
We save the file under the name **hello_world.py**. The suffix **.py** is important. It lets the system know that it deals with a Python file.

48% z Lekce 1



After saving, Atom should interpret the text as Python code - color the code. We can run the code with the **F5** or **F6** key.

48% z Lekce 1

A screenshot of a Python IDE window titled 'PythonBeginner'. The editor shows a file named 'hello_world.py' with two lines of code: '1 print("hello world")' and '2'. The first line is highlighted with a red box. Below the editor, a terminal window titled 'C:\Users\drpou\AppData\Local\Programs\Python\Python36\python.exe' shows the output 'hello world' (highlighted with a red box) and the message 'Process returned 0 (0x0) execution time : 0.088 s Press any key to continue . . .'. The status bar at the bottom indicates 'hello_world.py 1:21' and 'CRLF UTF-8 Python 0 files'.

Installing pip

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [INSTALLING PIP](#)

Pip stands for Python "Pip Installs Packages" or "Pip Installs Python" (yes it uses pip again). We will need pip in order to install new functionalities into our current version of Python. However, if you have Python Anaconda installed, then you do not have to install pip. Also pip is automatically included with Python since Python 3.4. Therefore if you have Python of version 3.4. or higher, you can skip this step.

Check if you already have pip

To find out, whether you have pip already installed on your computer, you can write the following command to the terminal window:

48% z Lekce 1

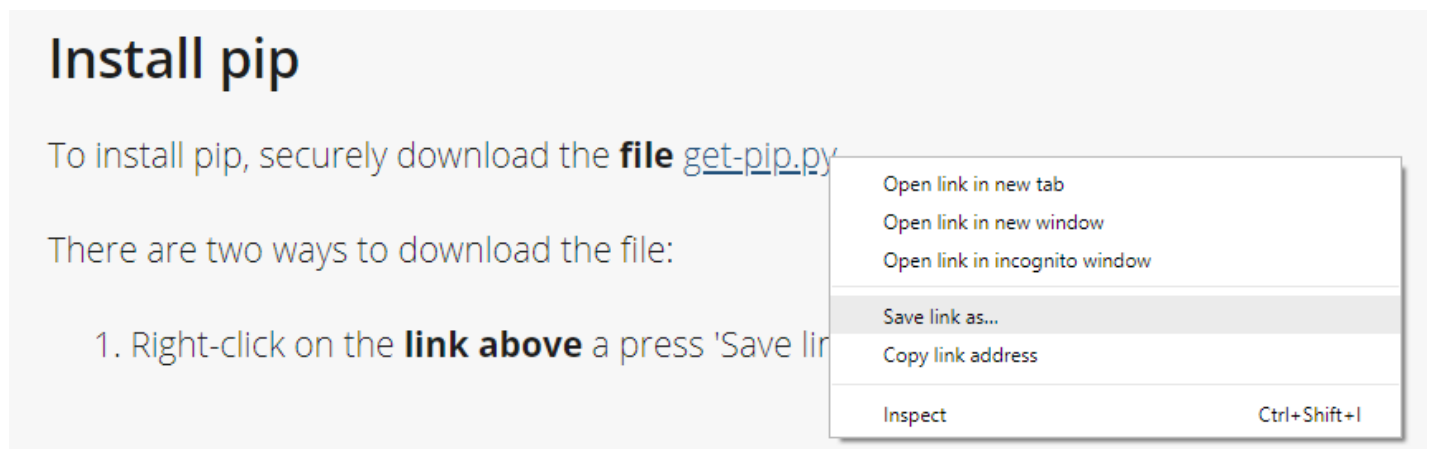
If no positive response is returned, you should install pip in the following way:

Install pip

To install pip, securely download the **file** [get-pip.py](#).

There are two ways to download the file:

1. Right-click on the **link above** and press 'Save link as...' and save the file.



2. Click through on the **page above**. You can ignore the code on the page. All you have to do there is to right-click and choose 'Save as...' and save the file.

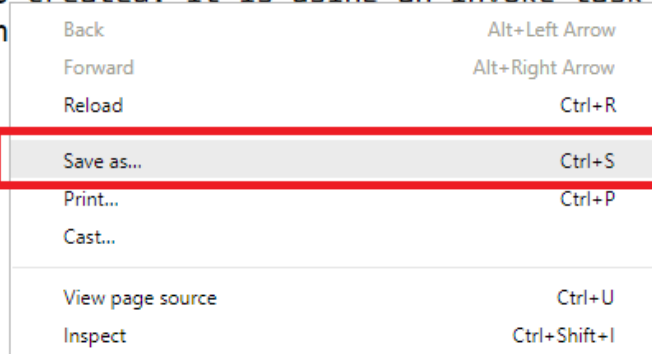
48% z Lekce 1

```
# You may be wondering what this giant blob of binary data here is, you might
# even be worried that we're up to something nefarious (good for you for being
# paranoid!). This is a base85 encoding of a zip file, this zip file contains
# an entire copy of pip (version 18.1).
#
# Pip is a thing that installs packages, pip itself is a package that someone
# might want to install, especially if they're looking to run this get-pip.py
# script. Pip has a lot of code to deal with the security of installing
# packages, various edge cases on various platforms, and other such sort of
# "tribal knowledge" that has been encoded in its code base. Because of this
# we basically include an entire copy of pip inside this blob. We do this
# because the alternatives are attempt to implement a "minipip" that probably
# doesn't do things correctly and has weird edge cases, or compress pip itself
# down into a single file.
#
# If you're wondering how this is created. it is using an invoke task located
# in tasks/generate.py called "in
# ``invoke generate.installer``.
```

```
import os.path
import pkgutil
import shutil
import sys
import struct
import tempfile
```

```
# Useful for very coarse version
PY2 = sys.version_info[0] == 2
PY3 = sys.version_info[0] == 3

if PY3:
    iterbytes = iter
else:
    def iterbytes(buf):
```



Open command line window and navigate to the folder containing get-pip.py and install it:

```
python get-pip.py
```

Installing packages with pip

Later on, when we will need to install some new functionality, we will just write the:

```
pip install <package_name>
```

The Flow of Working with Python

PYTHON ACADEMY / 1. INTRO TO PROGRAMMING / THE FLOW OF WORKING WITH PYTHON



We have installed Python, Editor and Pip. Also, we know the basics of commmand line. We are ready to learn, how to work with the Python code.

Editor + command line

In the following steps, we'll see how we could work with our code editor and command line together:

- 1. Create the working file 'PythonBeginner' in your home directory:

Operating System	Home Directory
Windows	C : \Users\<username>\PythonBeginner

48% z Lekce 1

Mac

`/Users/<username>/PythonBeginner`

2. Open your editor and in it a new file
3. Write the following text in the new file: `print('Hello Python')`
4. Save this file under the name **first.py** into the working file - PythonBeginner.
5. Open your command line:

Operating System	Starting cmd
Windows	Start > write 'cmd' > Command Prompt
Linux	Super > search Terminal
Mac	Applications > Utilities > Terminal

6. With command line get to your working directory (where your new Python file is located)

Operating System	Command
Windows	<code>cd C:\Users\ <username>\PythonBeginner</code>
Linux	<code>cd /home/<username>/PythonBeginner</code>
Mac	<code>cd /Users/<username>/PythonBeginner</code>

7. Run Python file by writing its name into the command line: `first.py`
8. The result should look like this:

Hello Python

Command line - interactive mode

Sometimes we need to quickly try what does certain Python expression do. For this it's better to work only with command line in the interactive mode.

48% z Lekce 1

Windows

```
> python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+1
2
>>> quit()
>
```

Unix (Linux/Mac)

```
$ python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+1
2
>>> quit()
$
```

Code blocks

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [CODE BLOCKS](#)

In the context of our course, we have many code examples. Let's call them code blocks, will be depicted in two ways:

1. Python Interactive Session

Lines in this code block will start with the `>>>` prompt at the beginning of line

```
>>> print('Hello')
```

48% z Lekce 1

begin with the `>>>` prompt.

Once we have written our code and pressed enter, Python executes the code and prints the result of the operation to the **output** which is again our terminal.

```
>>> 5 + 5
10
```

2. Python script

Python scripts will not contain the `>>>` prompt. You should write this code using your **code editor** into a file, which you will save as `.py` file.

```
1 print('Hello Python')
```

Code that will imply running the python script **within the editor** will simply be in the black code block:

```
'Hello Python'
```

Code that will imply running the python script from **command line** will begin with the dollar sign `$` or with comparison operator `>`:

```
$ python hello_python.py
'Hello Python'
```

This means, we want you to run the script named `hello_python.py`.

Online Python Editor

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONLINE PYTHON EDITOR](#)

```
1 # In our courses you can work with this online code editor
2
3 # We recommend you also work in your own terminal and editor because
```

48% z Lekce 1

```
8 # Also, after finishing this course, you'll be glad to know how to work with the
9
10 # Btw. using '#' at the beginning of the line signifies so call 'comment' and is
11
12 # Try is for yourself, run the code! :)
13 print('See? Only this is taken into account.')
14
```

spustit kód

48% z Lekce 1

Our Task For This Lesson

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONSITE PROJECT](#) / [OUR TASK FOR THIS LESSON](#)

We were asked to create a program that will allow user to reserve a trip to a given destination. We offer 6 destinations for the following prices:

Destination	Price
1 - Prague	1000
2 - Wien	1100
3 - Brno	2000
4 - Svitavy	1500
5 - Zlin	2300
6 - Ostrava	3400

Also, for destinations **Svitavy** and **Ostrava** we offer a special **discount of 25%**.

Once the user selects the destination, we want our program to calculate the price and subsequently ask the user for registration collecting the following data:

- name, surname, year of birth, email & password

To be continued in lesson #2

However, there are few constraints on the format, we want the collected data to be in:

- We cannot provide our services to clients under 15 years of age
- Emails have to contain @ symbol

48% z Lekce 1

If at least one of the above requirements is not met, the whole reservation process should be cancelled with appropriate message.

On the other hand, if everything went well the program should summarize the reservation details mentioning the user's name, destination and price.



How would we do that?

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONSITE PROJECT](#) / [HOW WOULD WE DO THAT?](#)

The best way, how to solve a programming problem is to:

1. Think of it as if you - human - would have to fulfill all the tasks, not the computer. How would you do that?
2. Divide the whole assignment into smaller tasks. This strategy is called **DIVIDE & CONQUER**

48% z Lekce 1



Once we know, what has to be done (point 1), we can divide the assignment into small tasks, we will gradually fulfill.

So what has to be done?

1. Communicate with the client -> write something to the screen
2. Take information from the client -> we call this inputs
3. Calculate the price
4. Check the validity of inputs, provided by the user

What tasks in what order will we do?

1. First, our program should be able to write at least something to our screen, so we know it is alive. Let's greet our users
2. Then we ask for the destination
3. Check whether the destination is valid
4. Calculate the price

Once we know how to do these four things, we can perform the rest as well.

48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [GREETING ROBOT](#) / [WRITING ON THE SCREEN](#)

To write anything we want to the screen, we need to use the command `print()`. The content we want to write to the screen goes inside the parentheses following the command name:

`print(stuff to be printed)`



So what will we print?

Today we will print:

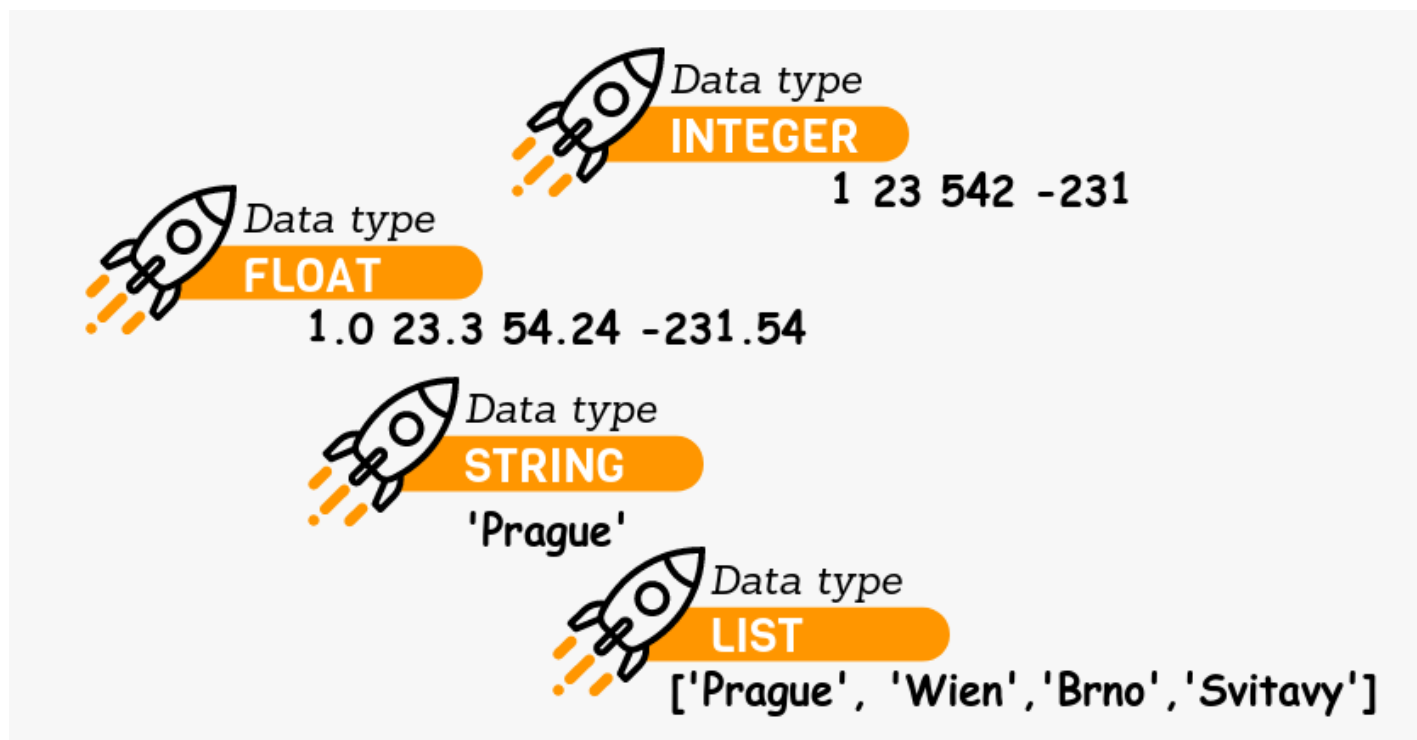
1. Numbers (34, 5.16, 3146, 43.741)
2. Text - strings - 'Hello World'
3. Lists

Let's see how each of them look like:

```
1  # This is comment and it is ignored
2
3  # Below we are printing numbers
4
5  # These are called integers
6  print(12)
7  print(645)
8  print(0)
9
10 # These are called floats - note the dot
11 print(12.2)
12 print(0.0)
```

48% z Lekce 1

```
16 print('Prague, Wien, Brno, Svitavy, Zlin, Ostrava',  
17  
18 # Now we are printing lists  
19  
20 print(['Prague', 'Wien', 'Brno', 'Svitavy', 'Zlin', 'Ostrava']))
```

[spustit kód](#)

We Should Store All Our Data

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONSITE PROJECT](#) / [WE SHOULD STORE ALL OUR DATA](#)

48% z Lekce 1

That somewhere is called **variable**. Variable is just a name, we give to a value, in order we can refer to it in the future - later in our program.

So the two operations, we perform with variables is:

1. We store the information in them
2. We retrieve information from it

You can try it in the editor below.

```
1 # 1
2 name = 'Martin'
3
4 # 2
5 print(name)
```

spustit kód

48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [CHALLENGE QUEST](#) / [FIRST PART OF OUR PROGRAM](#)

Now we know, that

1. We can represent the information in our program using numbers, strings and lists.
2. We can store their individual values in variable for later retrieval
3. We can print all that information using `print()` command

We can therefore begin to work on our today's program. We want to:

1. Greet the user once the program is turned on
2. Print the current offer of the destinations
3. Store the destination names in a variable
4. Store the destination prices in another variable
5. Store the destinations with discount again in a separate variable
6. Print all the variable created in order we know, that everything works fine

Our destinations with prices are:

Destination	Price
1 - Prague	1000
2 - Wien	1100
3 - Brno	2000
4 - Svitavy	1500
5 - Zlin	2300
6 - Ostrava	3400

Our destinations with 25% discount are **Svitavy** and **Ostrava**.

1 |

48% z Lekce 1

[spustit kód](#)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

[Click to see our solution](#)

Calculations

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONSITE PROJECT](#) / [CALCULATIONS](#)

This is something we already probably know from the elementary school - we can perform arithmetic operations with numbers. Those operations, we will be interested in for now are:

48% z Lekce 1

-	subtraction	10 - 4	6
/	classic division	10 / 4	2.5
*	multiplication	10 * 4	40
**	exponentiation	10 ** 4	10000

Also there are other important but maybe less known operators for the beginners:

Operator	Name	Use	Result
//	floor division	18 // 4	4
%	remainder	18 % 4	2

Let's try to experiment with it a bit. Try to use all the operators on the variable below and print the results of each operation.

```
1 a = 5
2 b = 2
```

spustit kód

Math with strings?

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONSITE PROJECT](#) / [MATH WITH STRINGS?](#)

We can use two arithmetic operators on strings and lists - addition `+` and multiplication `*`. There is however a small difference in the result compared to use with numbers.

Operation of using `+` on strings and lists is called **concatenation**.

```
1 print('Hello' + ' ' + 'World')
```

Operation of using `*` on strings and lists is called **repetition**. An integer value has to follow after the star symbol.

```
1 print('Hello' * 3)
```

Try to use these two operators on the variables below.

```
1 a = 20
2 b = '='
3 l1 = [1,2]
4 l2 = [3,4]
```

48% z Lekce 1

[spustit kód](#)

Comparisons

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONSITE PROJECT](#) / [COMPARISONS](#)

Another type of operations we may know from the elementary school, are comparison operations. Two values can be compared as follows:

Operation	Notation
A smaller than B	A < B
A greater than B	A > B
A smaller or equal to B	A <= B
A greater or equal to B	A >= B
A equal to B	A == B
A not equal to B	A != B

We will need these operations to find out, whether the person's age is above 15 as well as to check the length of various strings.

Try to compare the values below, what will happen?

```
1 num1 = 5
2 num2 = 2
3 text1 = 'a'
4 text2 = 'e'
5 list1 = [1,2,3]
```

48% z Lekce 1

[spustit kód](#)

Cutting Strings & Lists

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONSITE PROJECT](#) / [CUTTING STRINGS & LISTS](#)

In our program the user gives us the number of selected destination, not its name. Now we will learn why would we need such a number.

Letters in strings and items in lists are ordered. This is true, because, the meaning of words would be different if the order of letters changed. We can use this fact in our program, once the user provides us the number of the selected destination. Based on that number we can find the selected destination in our list. We say, that it is the destinations **index** (position).

To get the selected item from the list (or string), we will need to place it in the square brackets following the list or the string. For example we would like to select string **'apple'** from the list below - so we place the number 3 into the square brackets following the **fruits** variable name.

48% z Lekce 1



```
1 fruits = ['banana', 'peach', 'apple', 'orange']  
2 print(fruits[3])
```

[spustit kód](#)

48% z Lekce 1

Position numbering (indexing) **starts from the number 0**, not from the number 1. Therefore the first item in the sequence (string or list) will be at index 0.

How will we get the last item from the sequence? Try to play with negative indices.

String	H	E	L	L	O
Forward	0	1	2	3	4
Backward	-5	-4	-3	-2	-1

Second part of our program

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONSITE PROJECT](#) / [SECOND PART OF OUR PROGRAM](#)

Now we should be able to:

1. cut the destination name and price from our lists - we would like to take destination number 4
2. calculate the price for destinations with 25% discount
3. Print **'We have made your reservation to'** and the destination name
4. print the resulting price to the screen

```

1  # greet the client
2
3  print('''Welcome to the DESTINATIO,
4  place where people plan their trips''')
5
6
7  # offer destinations
8  print('We can offer you the following destinations:')
9
10 print('''
11 1 - Prague   | 1000
12 2 - Wien    | 1100
13 3 - Brno    | 2000
14 4 - Svitavy | 1500

```

48% z Lekce 1

```
18
19
20 DESTINATIONS = ['Prague', 'Wien', 'Brno', 'Svitavy', 'Zlin', 'Ostrava']
21 PRICES = [1000, 1100, 2000, 1500, 2300, 3400]
22 DISCOUNT_25 = ['Svitavy', 'Ostrava']
23
24 print(DESTINATIONS)
```

[spustit kód](#)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

[Click to see our solution](#)

Changing Data Types

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONSITE PROJECT](#) / [CHANGING DATA TYPES](#)

If you have tried to print the price with some comment prepended in front of it, you have probably end up with error. This is because, Python does not know, how to perform certain operations if mixing different data types. One example is joining of numbers and strings or lists and strings etc.

In order we can add the price to our string, we need to first **convert** the price from float or integer into a string. For that we will need to use the command `str()`. In order to keep it complete we list here all the conversion commands for the data types we know at the moment:

48% z Lekce 1

`str()`

a string - works with all the data types

`int()`

Converts the value inside the parentheses into an integer - does not work always

`float()`

Converts the value inside the parentheses into a float - does not work always

`list()`

Converts the value inside the parentheses into a list - does not work always

Let's do some conversion experiments with the values in our editor

```
1 num1 = 543
2 num2 = 56.12
3 text = 'Hello'
4 list = [text, num1, num2]
```

[spustit kód](#)

48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [CREATE PROJECT](#) / [SO WHAT TYPE IS THIS?](#)

After changing so many values into values of other data types, it would be helpful to have a tool, that will tell us, what type of value do we have stored in a given variable.

This tool is the command `type()`. We can pass it any Python value into its parentheses and it will tell us, what type of value it is.

Let's try it out. Print to the screen the types of values stored in variable `a`, `b`, `c` or `d` below:

```
1 a = '421'  
2 b = 421  
3 c = 421.0  
4 d = [a,b,c]
```

[spustit kód](#)

48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [CREATE PROJECT](#) / [GETTING THE USER INPUT](#)

We will need to collect many data from our user during the run of our program. Finally we are going to learn, how we can do that.

To get an input from an user, we need to use the command `input()`. Inside the parenthesis we write the string, which will be printed to the screen to prompt our user for the desired information. Therefore the string has to be well formulated. If we assign the command `input()` to a variable, the user response will be stored in that variable.

So to get an user's age, we can write the following line of code:

```
1 age = input('What is your age: ')
```

We should better try it in a terminal on our computer, because here, in the browser, the information is not collected the same way as in the terminal.

What type of value will be stored in the variable `age` ?

```
1 |
```

Third Part of Our Program

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [ONSITE PROJECT](#) / [THIRD PART OF OUR PROGRAM](#)

Now it is time

1. to collect all the information about the user:
 - destination number
 - name
 - surname
 - year of birth
 - email
 - password
2. print the thank you message with the user's name
3. print the message with the reservation price introduced by a string **'The total price is'**
4. also, it would be nice to add all those lines separating different sections of the program
(===== or -----)

As we are collecting the destination number, and we do not know, how to determine, whether a selected destination is among those with discount, we will skip calculating the discount price and we will just retrieve the normal price.

When running, our program could look like this:

```
=====
Welcome to the DESTINATIO,
```

48% z Lekce 1

```
we can offer you the following destinations:
```

```
-----
-----
```

```
1 - Prague   | 1000
2 - Wien     | 1100
3 - Brno     | 2000
4 - Svitavy  | 1500
5 - Zlin     | 2300
6 - Ostrava  | 3400
```

```
-----
-----
```

```
Please enter the destination number to select: 4
```

```
=====
REGISTRATION:
```

```
-----
-----
```

```
In order to complete your reservations, please share few details about
yourself with us.
```

```
-----
-----
```

```
NAME: Bob
```

```
=====
```

```
SURNAME: Rob
```

```
=====
```

```
YEAR of BIRTH: 2000
```

```
=====
```

```
EMAIL: bob@rob.com
```

```
=====
```

```
PASSWORD: password123
```

```
=====
```

```
Thank you Bob
```

```
We have made your reservation to Svitavy
```

```
The total price is 1500
```

```
1 # Greet the client
2
```

48% z Lekce 1

```
6
7 # Offer destinations
8 print('We can offer you the following destinations:')
9
10 print('''
11 1 - Prague | 1000
12 2 - Wien | 1100
13 3 - Brno | 2000
14 4 - Svitavy | 1500
15 5 - Zlin | 2300
16 6 - Ostrava | 3400
17 ''')
18
19 selection =
20 DESTINATIONS = ['Prague', 'Wien', 'Brno', 'Svitavy', 'Zlin', 'Ostrava']
21 PRICES = [1000, 1100, 2000, 1500, 2300, 3400]
22 DISCOUNT_25 = ['Svitavy', 'Ostrava']
23
24
25 destination = DESTINATIONS[selection - 1]
```

[spustit kód](#)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

[Click to see our solution](#)

Input & Output

PYTHON ACADEMY / 1. INTRO TO PROGRAMMING / BASICS / INPUT & OUTPUT

Throughout this course we will be learning not only how to write Python code, but especially general programming concepts, processes and workflows. The first important idea is the input and output cycle. When thinking about our programs in the future, always try to think of them in terms of inputs (what enters) and outputs (what comes out).

Input - Output Cycle

In order our programs to be useful, they will have to be able to accept or retrieve some data. We will learn about different sources of data and different ways how to retrieve or provide the data. For now, the most important is to realize, that programs retrieve data. Program then does its magic, and processes the data it has retrieved. This is the actual work, we designed the program to do for us. At the end, the program can, but does not have to provide some output. Any output can later serve as input to another process. That way, the imaginary cycle closes. If not used in another process, the output information can also be stored for the future use.

48% z Lekce 1



Input + Output = Data

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [BASICS](#) / [INPUT + OUTPUT = DATA](#)

Input and output are pretty general and abstract words. Behind the both terms we can imagine actual information - data. This information is represented in every programming language by various so called **data types** (numbers, text, sequences of numbers or text etc.). We will learn about those soon.

When speaking about input, we will ask ourselves, how the input data should be fed into the program. In other words, how they should be retrieved. Therefore **input implies retrieval of information**.

Programs that are designed to return some kind of processed information are expected to store it for the future use. For example, programs that calculate probability of Earth being hit by asteroid need to store the result somewhere in order humans can read it. Therefore **output implies storage of information**.

Programs use to chain the processing tasks, where one task takes some data in, processes it and outputs information, that is subsequently used by another processing task for further manipulation.

Demo

To better demonstrate what do we actually mean under input and output, let's do it with `print()` and `input()` functions.

The task of the `print()` function is to, well, print information to so called **standard output**. Standard output is in our case the terminal. We can write a number or text (function's input) into

48% z Lekce 1

```
>>> print(1234)
1234
>>> print('Hello')
'Hello'
```

In some way `input()` is similar to `print` - we can insert a string into parentheses. This string is later printed to the terminal (or generally standard output). The function `input()` then waits for us to write something to the terminal. Then it takes what we have written into the terminal as input and prints it again back to the standard output:

```
>>> input('What is your name? ')
What is your name? Martin
'Martin'
```

Summary

- Programmers goal is to make computer do tasks - process the information.
- In order the information can be processed, the information has to be first retrieved. Therefore the question will be what kind of information and from what source will be fed into the program.
- The next programmer's task is to design actual program that processes the data.
- Once the processing has terminated and output has been returned, the programmer has to decide, whether and how to store the information returned.
- Throughout this course, we would like to teach you to think in terms of inputs and outputs. You will notice that this theme repeats itself many times in various programming concepts.

Data Types Overview

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [BASICS](#) / [DATA TYPES OVERVIEW](#)

There are different kinds of data in Python. We humans, we differ between numbers and text. Python knows numbers and text (strings) as well, but recognizes much more data types

48% z Lekce 1

value None. It is of type NoneType.

Data Type	Literal	Example
int	Digits	5
float	Digits separated by dot	5.0
string	Characters enclosed in quotes	'abc' or '1234' etc.
tuple	Objects separated by commas and enclosed in parenthesis	(1, 2, 3, 4) or ('a', 'b', 'b')
list	Object separated by commas and enclosed in square brackets	[5, 5, 5, 5, 5]
range	Range keyword followed by parentheses - inside them the beginning point of the sequence and the stop value that is not anymore included in the sequence	range(0,10)
bytes	Similar to string with difference of prepending b in front of the first quote	b'5'
bytearray	Bytearray keyword followed by parentheses with the object representation inside them	bytearray(21324)
dictionary	Pairs of objects - key value pairs separated by colon. Key value pairs are separated by comma and everything is enclosed in curly braces	{'a':1, 'b':2}
set	Objects separated by commas enclosed in curly braces	{'a', 'b', 'c'}

48% z Lekce 1

None type

assigned yet to the variable

None

The word **literal** is used to denote, how a specific data type is represented in written form - in our code. Therefore the literal of integer is digit **5** and literal of string **'5'** is a character 5 enclosed in quotes.

Object - Few Words

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [BASICS](#) / [OBJECT - FEW WORDS](#)

Everything that comes to life in Python is an object. Data types we described are represented also by object - integer object, float object etc. For example integer object is number 5. Float object would then be 5.0 and string object would be '5'. In the data type overview, we mention multiple times word constructor. This word belongs into the concept of objects.

Built-in

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [BASICS](#) / [BUILT-IN](#)

We will meet the word built-in quite often. It denotes the notion of everything that comes with Python installation.

When you install Python on your machine, you are getting actually a package of things. Among others, there are:

- **Python interpreter** - program that knows how to interpret and run Python code we write
- **Standard library** - python files full of useful code that other developers can reuse

So everything that already comes packed inside the Python language is called built-in.

48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [BASICS](#) / [PYTHON LIBRARY](#)

Besides learning, how Python code is written, we will be learning how to use features already conveniently precoded for us in so called libraries. There is so to say, built-in library, called Python Standard Library that is included in Python standard installation. Then there are libraries created by third parties - these have to be installed additionally.

Python library is composed of so called **modules** what are nothing more than .py files. If there are more modules together, they create **packages**.

How can we imagine such a module?

Modules are files that contain Python code. The code inside them is logically related - its functionality focuses on solving similar problems. For example, soon we will meet math module. This module contains functionality for calculating factorial of a specific number, logarithms, greatest common divisors of two numbers etc.

We can access module's functionality by writing so called **import** statement into our code. In case of math, this would look like this:

```
import math
```

So when you will see this statement, you can be sure, that somebody is trying to access some Python code from external file, in order the code does not have to be repeatedly rewritten.

Storing Data

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [DATA](#) / [STORING DATA](#)

We know, that data on our computers is stored in files, databases etc. But the first level of storing the data is during the runtime of a program. We learned, that in order data - in Python objects - can come to exist, it has to be created. Once it is created, we need to store it somewhere, otherwise we cannot reference it. When we introduced notion of constructor, we played with them creating new objects. These were printed to our terminal, but that was all we could do.

So the next step for us is to store the objects during the program execution. We will store our data inside so called **variable**.

Variable

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [DATA](#) / [VARIABLE](#)

Variable can be imagined as a kind of a name for data we want to process. For example under the variable **age** we can store the number of years of a person. We store the data inside a variable using the assignment statement. That is, assigning a value to a variable. This is how **assignment statement** is written in Python:

```
>>> age = 32
```

Anytime we will want to get the age of that person for processing, we will just call the variable age.

48% z Lekce 1

Also, while we will be processing our data, the age of the person can change and so the number stored under the age variable can change.

```
>>> age = age + 1
>>> age
33
```

This changing nature is what gave the variable its name. We can repeatedly store in it varying values.

Assignment Statement

You have probably guessed, that assignment statement is in other words putting **equals sign =** between the variable name on the left side and the value on the right side. Assignment statement associates variable name with the assigned value.

What happens when we assign a value to a variable? Variable does not store the actual value assigned. It rather stores a **reference** to that value in memory. Referencing can be imagined as a line joining the variable name with the actual value. Variable is thus aware where to go for the value in the memory, if requested.

When Python sees assignment operation inside our code, it performs among others these **steps**:

1. The variable is created in computer's memory
2. The actual assigned value is created in memory
3. The variable reference to the value is created

It is not possible to use the variable name **before** it was assigned a value!! Variables that have not been assigned a value do not exist. It is not possible to declare variables without assigning any value to them.

Code Task

You can do the following task (but any other task in the future) here, in our online code editor, in your command line, or in your editor.

48% z Lekce 1

name in the `print()` function,

3. on the next line write variable name `height` (with or without the `print()` function - depending on if you're in editor or command line),

4. run the code and see what happens

```
1 # Assign the string to the variable name
2
3
4 # Print variable name
5
6
7 # Print variable height
```

spustit kód

You should get so called `NameError`. You get this error when we try to retrieve a value from a variable that does not exist. Python tries to look for variable `height` in the memory, but cannot find it and so returns the error.

48% z Lekce 1

can contain only:

- letters
- numbers
- underscores

and cannot:

- begin with a digit
- use so called reserved keywords (explained soon).

Examples of variable names:

Correct	Incorrect	Reason
maxSpeed	4every1	Number at the beginning
_maxSpeed	amountOf\$	\$ symbol not allowed
_maxSpeed1	36	Number at the beginning
NAME	'NAME'	' symbol not permitted
MaximumSpeed	Maximum Speed	Blank space not permitted

Use of the underscore symbol `_` as a name for a variable

Sometimes you will see just single underscore as a variable name. We can call such variables dummy variables. Values of these variables are actually not used later in the code. For example when performing sequence unpacking (which we talk more about later):

```
>>> a,_,b = ('Important','Not Important', 'Important')
>>> print(a,b)
Important Important
```

This rule is however only a practice of some programmers and not part or rule of Python language itself. We can call our dummy variables whatever we want.

48% z Lekce 1

variable's name in the code, it goes and looks for the value, we have originally assigned to it:

```
>>> name = 'Bob'
>>> name
'Bob'
```

Determining Data Type

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [DATA](#) / [DETERMINING DATA TYPE](#)

Python is dynamically typed language. This means that the same variable can be repeatedly assigned objects of different data types.

```
>>> my_var = 1
>>> my_var
1
>>> my_var = 'Bob'
>>> my_var
'Bob'
```

type()

In the example above we have assigned two different data types to the same variable without any problem. This is not possible in some languages. Being able to assign different data types to the same variable means that sometimes we will want to know what data type is there actually stored inside the variable. To find out what data type is referenced by the specific variable, we have to use **type()** function.

```
>>> type(my_var)
<class 'str'>
>>> my_var = 1
>>> type(my_var)
<class 'int'>
```

48% z Lekce 1

Code Task

This exercise should demonstrate the dynamic nature of Python - this feature will be very handy, when we will be programming. Your task is to assign the following values repeatedly to variable "my_var" and pass it to the `type()` function to find out what data types are these objects:

- 1
- 0.0
- 1,2,3
- 'abc'
- ['a',2,'b',1]
- {1,2,3}
- {1:'a',2:'b'}
- range(20)

Example:

```
>>> my_var = 1
>>> type(my_var)
<output>
```

```
1 # Variable
2 my_var = 1
3
4 # Calling the type() function
5 type(my_var)
```


48% z Lekce 1

[spustit kód](#)

Python is a Dynamic Language

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [DATA](#) / [PYTHON IS A DYNAMIC LANGUAGE](#)

So, what does 'dynamically typed language' means exactly? When created, variables in Python are not declared as type specific as in other languages - Java - meaning, that we can repeatedly assign objects of different data types to the same variable:

```
>>> a = 123
>>> a = 'abc'
>>> a = [4,5,6]
```

Dynamic typing allows for more flexibility by applying the same operation to different data types:

```
>>> def func(a,b):
>>>     print(a + b) #we do not know, what data type these variables will
reference -> the function can be used in more contexts
>>>
>>> a=43;b=52
>>> func(a,b)
95
>>>
>>> a='abcd';b='efg'
>>> func(a,b)
abcdefg
```

48% z Lekce 1

```
>>> tuple(a,b)
[1, 2, 3, 4, 5, 6]
```

The actual data type is determined (resolved) during the running of the program (runtime)

Strongly typed language

There are not automatic type conversions done by Python when 2 different data types meet in one operation:

```
>>> 'This is string' + 1234
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

First we have to convert int 1234 to a string in order we can perform concatenation

```
>>> 'This is string' + str(1234)
'This is string1234'
```

Constants

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [DATA](#) / [CONSTANTS](#)

Sometimes we will store in our variables values, that we do not want to be changed during the program. Such variables, whose values should not change are called constants.

It is agreed practice to write **constants names in upper case**:

```
1 WEEKDAYS = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
```

Reserved Keywords

48% z Lekce 1

Reserved keywords have special meaning for Python and when encountered in our code, Python triggers set of operations specific for each keyword. That means that each keyword has its purpose.

We cannot use reserved keywords for names of our variables. That is why they are called reserved. The table below should serve only for overview purposes and should create a notion that something like reserved keyword exists - please do not learn them by heart.

class	def	lambda	return	yield
try	except	finally	raise	assert
True	False	in	is	not
while	for	break	continue	
if	elif	else		
global	nonlocal			
and	or			
import	from			
pass	None			
with	as			
del				

Operations with Data Types

PYTHON ACADEMY / 1. INTRO TO PROGRAMMING / DATA TYPES / OPERATIONS WITH DATA TYPES

What distinguishes individual data types among them is the kind of operations we can perform on them. Numeric data types are used to calculate values, with strings and other sequences we benefit from the fact they are ordered somehow etc.

There are however also general principles shared among all data types and those are:

- **conversion** - we can sometimes convert a value of a given type into another data type
- **precedence** - operation precedence determines, which operation is executed first and which come later

Conversion

PYTHON ACADEMY / 1. INTRO TO PROGRAMMING / DATA TYPES / CONVERSION

Python does not implicitly convert data types in case needed for an operation (e.g. '3' + 3 is not 6, but error). Therefore if incompatible data types are combined in an operation, error is raised and the program execution is terminated. Tool used to convert data type in Python is data type constructor which "constructs" a new object of specified data type.

Here we have an overview of such constructors:

48% z Lekce 1

constructors: `int()` `float()` `str()` `list()` `tuple()` `range()` `bytes()` `bytearray()` `dict()` `set()`

Constructors are used not only to create (construct) a new object, but also to convert between different data type:

```
>>> num = 0.0
>>> int(num)
0
```

Above, we are converting value `0.0`, what is a float into integer `0`. We can also for example convert integer to string or the other way around:

```
>>> str(int())
'0'
```

```
>>> int('234')
234
```

However, every data type or object **cannot be converted** to every other data type or object. For instance we cannot convert set into an integer:

```
>>> int({1,2,3})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: int() argument must be a string, a bytes-like object or a
number, not 'set'
```

Try to guess

What would be the output in terminal when trying to convert string into an integer:

```
>>> my_var = '123'
>>> my_var = int(my_var)
>>> type(my_var)
```

Output

48% z Lekce 1

...trying to convert a list into an integer:

```
>>> my_var = [1,2,3]
>>> my_var = int(my_var)
>>> type(my_var)
```

Output ▼

...trying to convert string into a list:

```
>>> my_var = '123'
>>> my_var = list(my_var)
>>> type(my_var)
```

Output ▼

Code Task

- Store integer object of 25 in variable **a** ,
- then try to convert it to a float,
- finally find out the data type of the value stored inside **a**

```
1 # Store 25 in 'a'
2
3
4 # Convert it to float
5
6
7 # Find out the data type of 'a'|
```

48% z Lekce 1

[spustit kód](#)

Why only certain types can be converted?

Possibilities of conversion are limited by the characteristics of each data type. Collections of values (list, dictionary, set, tuple) cannot be converted into integer or float

Operation Precedence

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [DATA TYPES](#) / [OPERATION PRECEDENCE](#)

Sometimes, we will have multiple operations on one line of code. Python has to determine, what operations should be evaluated first. This is done based on operation precedence. Python contains rules according to which it determines the order of operations. We know this principle from school math, where if multiplication is present in the same expression with addition, first is performed multiplication, then addition.

Usually operations are performed from left to right if all the operators in an expression have the same precedence.

```
>>> 5 + 3 - 2 - 1
5
```

Otherwise those with higher precedence are evaluated first - in this case multiplication of **3 * 2** :

Data Type Specific Operations

PYTHON ACADEMY / 1. INTRO TO PROGRAMMING / DATA TYPES / DATA TYPE SPECIFIC OPERATIONS

Data types differ among themselves specifically by what **operations** we are able to perform with them. Integers are integers, because we can perform mathematical operations with them and strings are defined by their specific operations that no other data type can do. That means, each data type exists in Python, because each of them has its special superpowers and purpose.

But at the same time, different data types can also sometimes perform the **same operations** - with type specific result of course. This makes them more related.

Therefore we have numeric data types - we can perform common math operations, but some other operations are not possible. Or we have sequence data types - we can step through their items, etc.

We will explain data type operations in **4 separate units**:

1. Numeric data types operations (int, float, complex)
2. Sequence data types operations (str, tuple, list, range, bytes, bytearray)
3. Mapping data types operations (dict)
4. Set operations (set, frozenset)

Operations with Numeric Data Types

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [NUMERIC DATA TYPES](#) / [OPERATIONS WITH NUMERIC DATA TYPES](#)

There are 4 built-in numeric data types in Python. Each has its specific role and the way how it is written (its literal):

Abbreviation	Literal	Example
int	Whole numbers (no decimal point). There is not max value for int specified in Python. To simulate an integer value larger than any practical list or string index, we can use <code>sys.maxsize</code> .	123 , 2 , 0
bool	Represents boolean values - True or False. Value True is expressed by integer 1 and value False by integer 0. Therefore bool type is derived from int	True or False

float	numbers with decimal points are needed. Official Python documentation discusses the caveats of using float (it is not an exact representation of decimal number)	0.125, 3.14
complex	Number composed of two parts - real and imaginary (contains letter j). This data type is used in scientific calculations. We will not use this data type.	1.25 + 2j

This section should answer the questions, what can we actually do with numeric values (objects) in Python. We will learn about the following operations:

- arithmetic operations
- comparison operations
- bitwise operations (will be discussed later)

Arithmetic Operations

PYTHON ACADEMY / 1. INTRO TO PROGRAMMING / NUMERIC DATA TYPES / ARITHMETIC OPERATIONS

We have learned arithmetic in basic school. Python uses the same symbols to represent arithmetic operations. We first present the overview and then explain few terms that may not be so well-known.

Name	Syntax	Description	Output	Precedence	Example
------	--------	-------------	--------	------------	---------

48% z Lekce 1

addition	$a + b$	referenced by variable b is added to value referenced by variable a	integer	1	$5 + 6 = 11$
subtraction	$a - b$	value referenced by variable b is subtracted from the value referenced by variable a	integer	1	$5 - 6 = -1$
multiplication	$a * b$	value referenced by variable a is multiplied by value referenced by variable b	integer	2	$5 * 6 = 30$
classic division	a / b	value referenced by variable a is divided by the value referenced by variable b.	float	2	$5 / 6 = 0.8333333$

48% z Lekce 1

floor division	<code>a // b</code>	referenced by variable a is divided by the value referenced by variable b and the decimal part is rounded down.	int	2	<code>5 // 6 = 0</code>
remainder	<code>a % b</code>	value referenced by variable a is divided by value referenced by variable b and returned is the remainder.	integer	2	<code>5 % 6 = 5</code>
negation	<code>-a</code>	value referenced by variable a is changed to negative counterpart if originally positive and vice versa.	integer or float	3	<code>-6</code> or <code>-6.5</code>

48% z Lekce 1

power	<code>a ** b</code>	referenced by variable a is raised to power of the value referenced by variable b	integer	4	<code>2 ** 3 = 8</code>
-------	---------------------	---	---------	---	-------------------------

Note that the column precedence should be understood as a score system - highest score (4) highest precedence - will be evaluated first.

So for example:

- power will be always evaluated first,
- addition and subtraction always last.

Floor division

Floor division **returns always an integer** even though the classic division as we know it from mathematics would return a floating point number. The working principle is simple:

- If the classic division would return an integer, return this integer
- If the classic division would return a floating point number, round the number down (therefore the floor division) to the nearest integer and return this integer

Here we have the classic division:

```
>>> 5/6
0.8333333333333334
```

But if the same numbers (operands) are applied to the floor division operation, we get different value - rounded down - to the floor.

```
>>> 5//6
0
```

48% z Lekce 1

```
-1
```

Truncation

Truncation **chops off** the floating part of the number and returns an integer. Truncation takes place when float data type is converted to int data type for example. Truncation behaves differently from floor division when applied to negative floats. Meanwhile floor division rounds down, truncation just chops off the decimal part of the float.

```
>>> int(5/6)
0
```

```
>>> int(-5/6)
0
```

We mention here truncation, because it can seem similar to floor division at first, but it is a different operation.

Operator Precedence

Arithmetic operations are in Python evaluated from left to right and based on operator precedence. Operator precedence determines, which operations are evaluated **first** in an expression. First are evaluated operations with the highest precedence (as listed in the table at the beginning of the section) and subsequently those with lower.

In an expression: $a + b * c ** d$

1. first is evaluated $c ** d$
2. second is evaluated $b *$ and the result of $c ** d$
3. lastly is evaluated $a +$ the result of $b * c ** d$

However, we can influence the evaluation order using **parentheses**. The most nested expressions are evaluated first.

In an expression: $(a + (b * c)) ** d$

48% z Lekce 1

3. lastly is evaluated the result of `(a + (b * c))` and `** d`

Comparison Operations

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [NUMERIC DATA TYPES](#) / [COMPARISON OPERATIONS](#)

Result of comparison operations is always a boolean value - `True` or `False`. Comparison operations performed with numeric data types answer questions, whether one value is greater, smaller or (not) equal or identical to the other one. Expressions, where less than and greater than operations are included can be performed only when on **both sides** there are numeric values or values of the same data type.

Name	Syntax
less than	<code>a < b</code>
less than or equal to	<code>a <= b</code>
greater than	<code>a > b</code>
greater than or equal to	<code>a >= b</code>
equal to	<code>a == b</code>
not equal to	<code>a != b</code>
identical	<code>a is b</code>
not identical	<code>a is not b</code>

Logical Operations

Comparison operators can be connected into more complex statements using logical operators `and` or `or`. These operations are explained in a separate module.

Chained comparison operations

48% z Lekce 1

the more readable, then comes the `and` chain comparison with the expression `a < b < c`.

Now this is much more readable :)

On the other hand, chaining cannot be used always because sometimes it is less intuitive for the reader. What would be the result of the below expression?: `a == b > d`

Code Task

What will happen if you try to compare numeric data type with sequence, set or mapping?

```
>>> a = 123
>>> b = 'a'
>>> a < b
```

Output

What about identity comparison?

```
>>> a is b
>>> a != b
```

Output

Built-in Functions

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [NUMERIC DATA TYPES](#) / [BUILT-IN FUNCTIONS](#)

Here are some built-in functions that accept numeric input and return numeric data type:

48% z Lekce 1

<code>abs(number)</code>	float,int or complex	Data type corresponds to the input type.	Returns absolute value or magnitude of input value	<code>abs(-12)</code> <code>abs(12.0)</code>	12 12.0
<code>round(number[,ndigits rounding to])</code>	float or int	Data type corresponds to the input type	If input is float, the function returns the floating point value number rounded to ndigits after the decimal point. If ndigits is omitted or is None, it returns the nearest integer to its input.	<code>round(-12)</code> <code>round(12.12,1)</code> <code>round(12.12)</code>	-12 12.1 12
<code>divmod(number1, number2)</code>	Noncomplex number	data type corresponds to the input type	Returns a tuple of (number1 // number2, number1 % number2) - result of floor division and remainder	<code>divmod(7,3)</code>	(2,1)

You can find more info in [Python Standard Library Documentation](https://docs.python.org/3/library/).

48% z Lekce 1

and return only specified types of values.

1. What is the absolute value of: string `'name'` , tuple `(23, 'a', 321)` , list `[213, 32]` , dictionary `{'name': 'John', 'age': 32}` , set `{'a', 'b', 'c'}`
2. Try to round the value of string `'first name'`
3. What is the result of floor division of number `23452` by number `35` and what is the remainder?

```
1 # Absolute values
2
3
4
5
6 # Rounding the string
7
8
9 # Floor division|
```

spustit kód

Code Solution

48% z Lekce 1

[Click to see our solution](#)

Boolean Definition

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [BOOLEANS](#) / [BOOLEAN DEFINITION](#)

Boolean data type, also called **bool**, is considered numeric data type. We dedicate to bool a separate section, because it has its specific purpose different from that of ints, floats and complex numbers. It is not intended for mathematical calculations but for determination of so called **object truth value**.

48% z Lekce 1

They are created using `bool()` constructor.

- The `True` value is equivalent for numerical value of 1
- The `False` values is equivalent for value 0

That is why boolean data type is a subtype of integer - it covers only values 1 or 0.

To see how `bool()` behaves try to write expressions `bool(1)` and `bool(0)` into the terminal. You should get something like this:

```
>>> bool(1)
True
>>> bool(0)
False
```

Arithmetics with Booleans

As boolean data type is a **subtype of integer**, we can perform arithmetic as well as comparison operations with them.

Boolean data type's primary usage is not related to complicated arithmetics, however, we can make use of the fact, that arithmetic operations are also possible on bools.

We will appreciate this fact, when we will have a collection of items - let's say numbers, and we will want to find out, whether all of them have zero value. In other words, if all the items in the sequence have boolean value `False`.

The sequence can be composed of variables that represent numeric values:

```
>>> my_list = [0, 0, 0, 0]
```

To find out, whether all values referenced from the list, we could apply a built-in function `sum()` and determine, whether it evaluates to 0:

```
>>> sum(my_list)
0
```

48% z Lekce 1

operation as the following one is absolutely plausible:

```
>>> True > False
True
>>> False > True
False
>>> False == True
False
```

And now let's go to see, what are booleans usually used for.

Testing Truth Value of Objects

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [BOOLEANS](#) / [TESTING TRUTH VALUE OF OBJECTS](#)

If instead of entering zero or one into the `bool()` constructor, we would insert any other object (value), we would get returned its truth (boolean) value.

```
>>> bool('abc')
True
>>> bool([1,2,3])
True
>>> bool('')
False
```

Inside Python language, it has been defined, what Python should evaluate to **False** and what should be considered of **True** value.

These values are considered **False** in Python:

- **None**
- **False**
- zero of any numeric type, for example, **0**, **0.0**, **0j**.

48% z Lekce 1

- instances of user-defined classes, if the class defines a `bool()` or `len()` method, when that method returns the integer zero or bool value `False` .

All other values are considered true — so objects of many types are always true.

You do not have to worry too much about the last point enumerated. This will be made clearer once we know, what a class and method is.

Code Task

Before you execute the following statements try to guess what will be the results.

```
1 print(bool(''))
2 print(bool('0'))
3 print(bool([0]))
4 print(bool({}))
5 print(bool('None'))
6 print(bool(0.1))
7 print(bool(['']))
```

spustit kód

48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [BOOLEANS](#) / [BOOLEAN OPERATIONS](#)

Now when we know, what values are considered True and False we can turn to evaluation of Boolean operations. The general rules rely on predicate logic taught at the high school. In the tables below, we will demonstrate the principles of boolean logic on two expressions glued together by **boolean operator** (and, or). Please note, that more than one boolean operator can be used in one expression even though we will use only one.

Boolean operations rely on predicate logic operators:

- and
- or
- not

Example of such an expression in use could be:

```
>>> a = True
>>> b = False
>>> a and b
False
```

The example above could lead us to a conclusion that the result of a boolean operation is a boolean. But that is not true. Especially when variables or expressions implied does not have to be only boolean values:

```
>>> a = 0
>>> b = 5 * a
>>> a and b
0
```

Value returned from a boolean expression depends on the truth (truthy) value of operands and also on boolean operator which glues them together.

Logical Operations

when working with and, or, not Python uses something called [short circuit evaluation](#). This process is basically supposed to speed up the evaluation of the 'truthiness' of the statement. Let's see in detail how each operand works.

AND

What is important about **and** operator is that Python already knows that the whole expression will be false if the **first part** of the expression evaluates to **False** (int 0 or empty string `' '` or empty list `[]` etc.). In such case Python stops the evaluation of the boolean expression there and returns whatever it finds in the first part of the expression. In other words, Python still returns value with boolean equivalent of **False** .

```
>>> a = 0
>>> b = 3.14
>>> a and b
0
```

First Operand Value	First Operand's Boolean Value	Operator	Second Operand Value	Second Operand's Boolean Value	Boolean Operation	The Result of the Boolean Operation	Explanation
a = 0	False	and	b = 5	True	a and b	0	First variable references int 0, which is considered False -> evaluation stops there and 0 is returned

48% z Lekce 1

Operand Value	Boolean Value	Operator	Operand Value	Boolean Value	Operation	the Boolean Operation	Explanation
a = 0	False	and	b = 0.0	False	a and b	0	The first expression is already considered False and therefore its actual value 0 is returned.
a = 0.0	False	and	b = 0	False	a and b	0.0	Both expressions are false - > still, it is enough to evaluate the first expression to know, that the whole statement is False. Therefore the value of the first expression is returned.

If the first part of the expression evaluates to **True** , then the second value - to the right of the

48% z Lekce 1

```
>>> a = 2
>>> b = 3.14
>>> a and b
3.14
```

```
>>> a = 2
>>> b = 0
>>> a and b
0
```

First Operand Value	First Operand's Boolean Value	Operator	Second Operand Value	Second Operands's Boolean Value	Boolean Operation	The Result of the Boolean Operation	Explanation
a = 342	True	and	b = 0	False	a and b	0	Boolean expression has to return the Falsy part

48% z Lekce 1

Operand Value	Boolean Value	Operator	Operand Value	Boolean Value	Operation	the Boolean Operation	Explanation
a = 2	True	and	b = 3.14	True	a and b	3.14	After evaluation number 2 to True, Python has to evaluate also the second expression and returns its value no matter what.

OR

We know that in case of **or** operator, Python can predict the result of the boolean operation **if the first expression (operand) evaluates to True**. In that case the whole boolean operation will evaluate to True. Python stops the evaluation of the boolean expression there and returns whatever value it finds in the first expression. This time the principle will be demonstrated using sequence data types.

```
>>> a = 2
>>> b = 3.14
>>> a or b
2
```

48% z Lekce 1

Operand Value	Boolean Value	Operator	Operand Value	Boolean Value	Operation	the Boolean Operation	Explanation
a = 'string'	True	or	b = []	False	a or b	'string'	In other words, return which ever is True
a = [1,2,3]	True	or	b = [3.14]	True	a or b	[1,2,3]	The first expression is already True, therefore it is returned.

The second operand's value is returned from the boolean operation, if the **first value** is considered **False**. It is because Python has to proceed with evaluating the second operand in order to determine, whether the Boolean **or** expression is **True**. Once moved to the second operand, there is no way back for returning the value of the first one. As in the example below:

```
>>> a = 0
>>> b = 3.14
>>> a or b
3.14
```

First Operand Value	First Operand's Boolean Value	Operator	Second Operand Value	Second Operand's Boolean Value	Boolean Operation	The Result of the Boolean Operation	Explanation
---------------------	-------------------------------	----------	----------------------	--------------------------------	-------------------	-------------------------------------	-------------

48% z Lekce 1

Operand Value	Boolean Value	Operator	Operand Value	Boolean Value	Operation	the Boolean Operation	Explanation
a = ()	False	or	b = 'abc'	True	a or b	'abc'	Python has to evaluate also the second expression to find out the result. In other words, return which ever is True

48% z Lekce 1

Operand Value	Boolean Value	Operator	Operand Value	Boolean Value	Operation	the Boolean Operation	Explanation
							After seeing that the first expression is False, Python still has to evaluate the
a = "	False	or	b = []	False	a or b	[]	second one to be sure, that the whole expression is False. Therefore the second expression's value is returned.

Use case

Short circuit evaluation can be also seen a kind of shorthand for conditional statement.

Short circuit evaluation for **and** operator (**var1 and var2**) can be seen as equivalent to:

```

1 if var1 == False:
2     return var1
3 else:
```

48% z Lekce 1

```
1 if var1 == True:
2     return var1
3 else:
4     return var2
```

Operator Precedence

As with arithmetic operators, some boolean operators have precedence before the rest - specifically **not** operator has precedence before **and** and **or** operator. **and** has precedence before **or**.

The below expression of **not having precedence** is evaluated in the following order:

1. first evaluated not c (evaluates to **True**)
2. then is evaluated result of not c and d (what evaluates to **False**).

```
>>> c = False
>>> d = False
>>> not c and d
False
```

The expression below - **changing the precedence using parentheses** is evaluated in the following order:

1. first evaluated c and d (evaluates to **False**)
2. then is evaluated negation of c and d (what evaluates to **True**).

```
>>> not (c and d)
True
```

In the example of or both operands in pairs c,d or in e,f have to be **True** in order the expression evaluates to **True**.

```
>>> e = True
>>> f = True
>>> (c and d) or (e and f)
```

48% z Lekce 1

as one from the pair d,e.

```
>>> c and (d or e) and f
False
```

Built-in Functions

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [BOOLEANS](#) / [BUILT-IN FUNCTIONS](#)

The functions below take as input objects like for example, list, string, tuples etc. - we call them iterables in short.

We will present here two built-in functions:

Function	Description	Input	Calling Function	Output
all()	all items of supplied sequence are of boolean value True	my_list = [1,2,3,4]	all(my_list)	True
		my_list = [0,1,2,3,4]	all(my_list)	False
		my_list = [1,2,3,4, '']	all(my_list)	False
any()	at least one of the items is True	my_list = [1,2,3,4]	any(my_list)	True
		my_list = [1,'',0, '']	any(my_list)	True
		my_list = ['',0, '']	any(my_list)	False
		my_list = ['',0, '']	any(my_list)	False
		my_list = []		

You can find more info on [all\(\)](#) and [any\(\)](#) in Python Standard Library Documentation.

48% z Lekce 1

Definition

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [DEFINITION](#)

Sequences are **ordered collections of data (objects)**. The word 'ordered' is very important, because it is this characteristic that makes them special and appropriate for specific kinds of tasks. Sequences support a specific set of operations - a specific protocol.

Table of sequence data types

Name	Description	Example
string	strings represent textual information - they have to be enclosed in quotes	'John Smith' , "Bob Marley"

48% z Lekce 1

tuple	of objects separated by commas - you will often hear that tuples cannot be changed once created	<code>('name', 'surname', 'age')</code> - tuple of strings
list	list is also an ordered collection of objects - but in contrast to tuple, we can change contents of a list object	<code>['name', 'surname', 'age']</code> - list of strings
range	range is a sequence of integers from start value to end value	<code>range(1,11)</code> represents values 1,2,3...,9,10
bytes	bytes represent binary data - in contrast to text, which has to be decoded into letters. bytes sequence cannot be changed	<code>b'bob marley'</code>
bytearray	bytearray is like bytes with difference that we can change the content of the bytearray object	<code>bytearray(b'bob marley')</code>

Operations overview

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [OPERATIONS OVERVIEW](#)

The operations we can do with sequences stem from their features.

Features of sequences

As the strings, tuples, lists etc., are classified into one group of sequence data types, they have to have some features in common. They even share some characteristics with other than sequence

48% z Lekce 1

The most important features are the fact that sequences are ordered, that elements can be...

- access individual items inside them by their position number (**indexing**)
- or select items in specific interval of positions (**slicing**),
- or select items at every n-th position (**striding**),
- or sort the items in the sequence (**sorting**)

We can also multiply the sequences (**repetition**).

Features in common with other collections

- sequences can be joined into longer sequences (**concatenation**)
- Sequences can contain multiple objects - therefore we can check their **lengths**
- we often want to know if some elements are present in the sequence (**membership test**)
- values in collections can be ordered based on their magnitude (**min or max**)

Indexing

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [INDEXING](#)

Indexing operation targets an item at a specific position. It returns the object at the designed position. Indexing operator in Python is square brackets immediately following the sequence object:

sequence[index]

In the example below, we are asking Python to return element at position 0 in the string 'Hello World'

```
>>> my_string = 'Hello World'
>>> my_string[0]
'H'
```

48% z Lekce 1

Here is an example, how positions are numbered in the example sequence - string 'Hello'

String	H	E	L	L	O
Forward	0	1	2	3	4
Backward	-5	-4	-3	-2	-1

Negative Index

We do not have to know the length of the sequence in order to retrieve the last item of the sequence. In Python, indexing can count also backwards from the end of the sequence. For this purpose are used negative integers, where the last item has position -1.

```
>>> seq = [0,1,2,3,4,5,6]
>>> seq[-2]
5
```

It is important to **note**, that the length of the sequence is +1 greater than is the value of the last index of that sequence:

```
>>> sequence = 'abcdef'
>>> len(sequence)
6
>>> sequence[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

If we try to access the item at position equal to the length of the sequence, we will get an error.

Indexing works with strings, tuples as well as lists.

Strings

48% z Lekce 1

Lists

```
>>> ['a', 'bc', 'd'][2]
'd'
```

Tuples

```
>>> ('a', 'bc', 'd')[2]
'd'
```

Slicing

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [SLICING](#)

Meanwhile indexing returns one item from the sequence, slicing returns a subsequence of items - a slice. To tell Python, where the subsequence should begin and end, two indices separated by the semicolon `:` are needed. Besides that, the indexing operator square brackets `[]` is still used as with indexing operation.

There are for ways we can perform slicing operation:

- `sequence[start:end]`
- `sequence[:end]`
- `sequence[start:]`
- `sequence[:]`

`sequence[start:end]`

The returned sequence will begin at index start and end at index end-1.

```
>>> '012345'[2:5]
'234'
```

48% z Lekce 1

The data type of resulting **subsequence** is the same as that of the original sequence:

```
>>> ['a','b','c','d','e'][1:2]
['b']
```

The slicing operation above returns only one item - at the position 1, however inside a list again - even though the item is a string object.

Note, how the tuple slice made of one item looks like - it contains comma following the item.

```
>>> ('a','b','c','d','e')[1:2]
('b',)
```

sequence[:end]

If start index is omitted, the subsequence is expected to **start at index 0**.

```
>>> [0,1,2,3,4,5][:4]
[0, 1, 2, 3]
```

sequence[start:]

If the end index is omitted, the subsequence will include all items from start index **till the end** of the sequence.

```
>>> [0,1,2,3,4,5][2:]
[2, 3, 4, 5]
```

sequence[:]

If both start and end index are omitted, the subsequence returned will be the **copy** of the original sliced sequence

```
>>> [0,1,2,3,4,5][:]
[0,1,2,3,4,5]
```

We can get the whole sequence also in the following way:

48% z Lekce 1

or:

```
>>> [0,1,2,3,4,5][0:6]
[0, 1, 2, 3, 4, 5]
```

In addition

If the slice starts and ends at the **same index**, empty sequence is returned. This is because the slice returns sequence from start till end - 1. In such case, last (and so the first item) is excluded from the subsequence.

```
>>> [0,1,2,3,4,5][:0]
[]
```

Striding

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [STRIDING](#)

Striding extends the slicing functionality by using **third value (step)** inside the square brackets. This third value tells how many items should be skipped when slicing. This step value is by default set to 1 (when not included).

The expression below returns every second item from the second till the eighth item in the sequence.

sequence[start:stop:step]

```
>>> '0123456789'[1:8:2]
'1357'
```

We can also return every second item from the whole sequence:

```
>>> '0123456789'[: :2]
'02468'
```

List Sort Method

`list.sort(key=None, reverse=False)`

This is an operation, that modifies the list in place. It changes the order of list items based on a criterion (key) specified as method argument. In order a sequence can be sorted, items it contains have to be comparable among themselves in terms of equality and inequality (using comparison operations).

Another argument can be specified - reverse. By default this is set to `False`, what means the sequence will be sorted in ascending order. If set to `True`, then the list will be sorted in descending order.

```
>>> lst = ['a',1,'3',''] #cannot order strings and ints
>>> lst.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
```

We mention the list `sort()` method here for completeness. For more details, see Python documentation:

- [Tutorial on sorting in Python](#)
- [Python documentation](#)

Sorting Immutable Sequences

`sorted(iterable, key=None, reverse=None)`

Besides indexing, slicing and striding, we can also sort items in immutable sequences with a built-in function `sorted()`. When speaking about immutable sequences, we mainly mean tuple and string as ranges are already ordered.

Takes the same inputs as `list.sort()` method, however `sorted()` is the function that has to be used with immutable sequences. The reason is that it returns a new list. On the other hand,

48% z Lekce 1

customize the sort order, and the reverse flag can be set to request the result in descending order.

Returning the new object vs in place operation is the main difference here compared to sorting mutable sequence like list. We can use sorted function on lists too. It will just return a new sorted list and will not change the original one.

And here is the prove, that the original list is not changed after applying `sorted()` function:

```
>>> a = [1,9,2,6,3,5,8]
>>> sorted(a)
[1, 2, 3, 5, 6, 8, 9]
>>> a
[1, 9, 2, 6, 3, 5, 8]
```

Repetition

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [REPETITION](#)

Repetition expression consists of a sequence that will be repeated, the multiplication operator `*` and an integer that specifies the number of repetitions.

`sequence * i`

In the example below we repeat the items of the list `[1,2,3]` three times. The order of items is maintained.

```
1 print([1,2,3] * 3)
```

Output:

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Repetition operation can be used in ASCII art for example - creating headers etc.:

```
1 print("=" * 40)
```

48% z Lekce 1

Output:

```
=====
Header
=====
```

The repetition operation can be performed on strings, lists, tuples:

Strings

```
>>> 'ab' * 2
'abab'
```

Lists

```
>>> ['ha'] * 3
['ha', 'ha', 'ha']
```

Tuples

```
>>> ('ha',) * 3
('ha', 'ha', 'ha')
```

Code Task

What will happen if you execute the following code?:

```
1 print('=' * 20)
2 print('hello' * 2 + 'world' * 2)
3 print(['a'] * 3 + ['b'])
4 print(('a') * 3 + ('b'))
5 print(('a',) * 3 + ('b',))
```

48% z Lekce 1

[spustit kód](#)

Concatenation

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [CONCATENATION](#)

Concatenation is operation that **merges two sequences into one**. It is performed when **+** operator is found between two sequences of the same data type.

```
>>> 'Hello' + ' ' + 'World'
'Hello World'
```

Important is to stress that it is not possible to concatenate 2 sequences of two different data types - e.g. string with list:

```
>>> 'a' + ['b']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'list' object to str implicitly
```

Concatenation works for strings, lists and tuples.

Strings

48% z Lekce 1

Lists

```
>>> [1,2,3] + [4,5,6]  
[1,2,3,4,5,6]
```

Tuples

```
>>> (1,2,3) + (4,5,6)  
(1,2,3,4,5,6)
```

Code Task

Try to concatenate:

- strings **'New'** and **'York'**
- lists **['New']** and **['York']**
- tuples **('New',)** and **('York',)**

```
1 # Strings  
2  
3  
4 # Lists  
5  
6  
7 # Tuples|
```

48% z Lekce 1



Length

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [LENGTH](#)

Length is special characteristic of sequences. The length of the sequence tells us, how many items it contains. We can use the built-in function `len()` to determine the number of items in a sequence:

```
>>> len([0,1,2,3,4,5])
6
```

The `len()` function works with strings, lists and tuples as well:

Strings

```
>>> len('abcfed')
6
```

Lists

```
>>> len(['a','b','c','d','e'])
5
```

Tuples

```
>>> len(('a','b','c','d','e'))
5
```

48% z Lekce 1

```
1 print(len(('abcde')))  
2 print(len(('abcde',)))  
3 print(len('First Name'))  
4 print(len([1,2,[3,4,5]]))
```

[spustit kód](#)

Membership testing

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [MEMBERSHIP TESTING](#)

To determine whether an item is present in a sequence (string, list, tuple) we use the **in** operator.

To find out, whether letter **'a'** is present in the string **'abc'**, we can write:

48% z Lekce 1

In the above code, we are asking: **Is 'a' present in 'abc'?**

The result of membership testing is **True** (the item is present) or **False** (item is not present).

Code Task

- First guess the result of the expressions below,
- then run the code,
- after uncomment one of the commented expressions,
- put back the hash and uncomment the second.

This way you should be able to see the results for all the expressions.

```
1 print('name' in 'first name')
2 print( 1 in [3,2,1])
3 print([3,2,1] in [3,2,1])
4
5 # print(1 in '12')
6 # print(1 in 12)
```

spustit kód

Max & Min

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [MAX & MIN](#)

Lastly we will learn how to get the maximum or the minimum of the sequence.

Determining max

For determining the maximum of a sequence we use the built-in function `max()`.

```
1 # List
2 my_list = [1, 2, 3, 4]
3
4 # Printing the maximum
5 print(max(my_list))
```

Output:

4

Determining min

Similarly, for determining the minimum of a sequence we use the built-in function `min()`.

```
1 # List
2 my_list = [1, 2, 3, 4]
3
4 # Printing the minimum
5 print(min(my_list))
```

Output:

1

48% z Lekce 1

```
2 my_list = ['Apple', 'Butter', 'Bread']
3
4 # Printing max and min
5 print(max(my_list))
6 print(min(my_list))
```

Output:

```
'Butter'
'Apple'
```

Why is 'Butter' the maximum and 'Apple' the minimum? Because the functions `max()` and `min()` compares the values of individual characters according to the ASCII table, where each character has its own value and also its ranking.

Note that when trying to get the max or min of different data types we get an error!:

```
1 # List
2 my_list = [1, 'Apple', 2, 'Butter', 3, 'Bread']
3
4 # Printing max
5 print(max(my_list))
```

Output:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
```

List Operations

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [LIST OPERATIONS](#)

Examples in the table below will simulate action on the list `lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

48% z Lekce 1

Item replacement	<code>seq[index] = value</code>	index is replaced by a new value.	<code>lst[2] = 75</code>	<code>[4, 5, 6, 7, 8, 9]</code>
Slice replacement	<code>seq[start:end] = iterable</code>	The slice and the iterable have to be the same length. If the iterable is smaller number than the sequence slice, the rest in the sequence slice is discarded. If the iterable is longer, than slice, all items from the iterable are inserted into the sequence.	<code>lst[3:6] = [65, 98]</code>	<code>[0, 1, 2, 65, 98, 6, 7, 8, 9]</code>
Item removal	<code>del seq[index]</code>	Removes item at specified index	<code>del lst[3]</code>	<code>[0, 1, 2, 4, 5, 6, 7, 8, 9]</code>
Slice removal	<code>del seq[start:end]</code> or <code>seq[start:end] = []</code>	Removes slice from item at index start till item at index end-1.	<code>del lst[3:6]</code>	<code>[0, 1, 2, 6, 7, 8, 9]</code>
Striding removal	<code>del seq[start:end:step]</code>	Removes every nth item in a slice spanning from index start till index end-1.	<code>del lst[::2]</code>	<code>[1, 3, 5, 7, 9]</code>

48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [LIST METHODS](#)

Examples in the table below will simulate action on the list `lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

The method are only functional with list - `lst.method(argument)`

Operation	Syntax	Description	Example	Output
Insertion	<code>.append(x)</code>	Appends 'x' value at the end of the list	<code>lst.append('50')</code>	<code>[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 50]</code>
Insertion	<code>.insert(i, x)</code>	Inserts 'x' value at the index 'i'	<code>lst.insert(0, 420)</code>	<code>[420, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]</code>
Insertion	<code>.extend(seq)</code>	Extends the list by another list	<code>lst.extend([420, 2510])</code>	<code>[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 420, 2510]</code>
Item Removal	<code>.remove(x)</code>	Removes item 'x' from the list. If 'x' is not found, we get an error	<code>lst.remove(9)</code>	<code>[0, 1, 2, 3, 4, 5, 6, 7, 8]</code>
Item Removal	<code>.pop([i])</code>	Returns 'i' and deletes it from the list. If 'i' is not stated, the method takes the last value of the list by default	<code>lst.pop()</code>	<code>9; [0, 1, 2, 3, 4, 5, 6, 7, 8]</code>

48% z Lekce 1

Item Removal	<code>.clear()</code>	items from the list. The method takes no arguments	<code>lst.clear()</code>	<code>[]</code>
Count	<code>.count(x)</code>	Returns count of 'x' in the list	<code>lst.count(5)</code>	<code>1</code>
Sort	<code>.sort(key=None, reverse=False)</code>	Returns sorted sequence according to key and the parameter <code>reverse</code>	<code>lst.sort(reverse=True)</code>	<code>[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]</code>
Search	<code>.index(x)</code>	Finds 'x' in list and returns its index	<code>lst.index(5)</code>	<code>5</code>
Reversion	<code>.reverse()</code>	Changes the order of the list. The method takes no arguments	<code>lst.reverse()</code>	<code>[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]</code>
Copy	<code>.copy()</code>	Returns a copy of the original list. The method takes no arguments	<code>lst_copy = lst.copy()</code>	<code>lst_copy = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]</code>

Strings

PYTHON ACADEMY / 1. INTRO TO PROGRAMMING / SEQUENCE DATA TYPES / STRINGS

Creation and Syntax

When speaking about string syntax, we are actually asking: how can we identify a string? In order Python can recognize a string, the sequence of characters has to be enclosed in quotation marks. There are 3 types of quotations that can be used:

- Single Quotes
- Double Quotes
- Triple Quotes

```
>>> single_quote = 'Hello World'
>>> double_quote = "Hello World"
>>> triple_quote = '''Hello
World'''
```

Single & Double Quotes

Single and double quotes are used interchangeably, meaning that there is no difference btw. a string written in single or double quotes. However, we have these two types of quotes for cases such as:

1. We want to write a quote inside the string.

We would use one type of quotes to enclose the string and the other type to write the quote inside the string

```
>>> quoted_str = 'He used to say: "Nothing is for free"' #single quotes
enclosing
>>> quoted_str = "He used to say: 'Nothing is for free'" #double quotes
enclosing
```

2. We want to use apostrophe inside the string.

We would have to use double quotes to enclose the string and then we can use an apostrophe

```
>>> apostr1 = "It's time for break" #apostrophe inside
```

48% z Lekce 1

Instead of combining single and double quotes, we could also use the backslash `\"` to so called "escape" the quote character inside the string. However, escaping does not look so elegant as combining quote types:

```
>>> apostr2 = 'It\'s time for break'
>>> apostr2
"It's time for break"
```

Triple quotes

Triple quotes are actually just 3 single or double quotes on each end of the string. Therefore, we should still pay attention to quoting and apostrophes inside the string. Special power of triple quotes is that they allow the text to spread across multiple lines - this is not possible with single or double quoted strings. Triple quotes are therefore used for longer texts, especially for the code documentation.

```
1 triple = '''This string will span over multiple lines and no error will
    be raised by Python interpreter.
2         Still, we should escape the apostrophes \' or use opposite
    quote type'''
```

What will happen if we try to use the same type of quote inside a string as the quote type enclosing the string? Try to find out by writing the below string to your terminal session. For example write the following:

```
>>> 'it's free'
```

String constructor

Besides enclosing text inside quotes, strings can be created with `str()` constructor:

```
>>> num_str = str(123)
>>> num_str
'123'
```

48% z Lekce 1

behind the scenes and therefore we can perform comparison operations as greater than or less than on strings.

Character's number representation can be acquired using `ord()` built-in function:

```
>>> ord('z')
122
```

```
>>> 'abc' < 'acb'
True
>>> 'aab' < 'aac'
True
```

The comparison is performed elementwise - comparing characters at the same index ('a' to 'a', 'a' to 'a', 'b' to 'c')

String Methods

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [SEQUENCE DATA TYPES](#) / [STRING METHODS](#)

We've created quite a list here, but it still doesn't include all the methods. For more info on string methods go to [Python documentation](#). Each method is preceded by a string - `'string'.method(argument)`

Operation	Syntax	Description	Example	Output
Letter change	<code>.lower()</code>	Return string with all letters lowercased	<code>'HA HA'.lower()</code>	<code>'ha ha'</code>
Letter change	<code>.upper()</code>	Return string with all letters uppercased	<code>'he he'.upper()</code>	<code>'HE HE'</code>

48% z Lekce 1

Letter check	<code>.islower()</code>	value <i>True</i> , if all items in string are lowercased. Otherwise returns <i>False</i>	<code>'David'.islower()</code>	<code>False</code>
Letter check	<code>.isupper()</code>	Returns boolean value <i>True</i> , if all items in string are uppercased. Otherwise returns <i>False</i>	<code>'PETR'.isupper()</code>	<code>True</code>
Letter change	<code>.title()</code>	Returns a new string, where every word begins with capital letter. If a number sequence precedes letters, the method will uppercase the first letter anyway. Also if there is apostrophe within the word, the letter succeeding the apostrophe is also capitalized.	<code>'123word'.title()</code>	<code>123Word</code>

48% z Lekce 1

Letter change	<code>.replace(what, for what)</code>	substring for another substring inside the operated string.	<code>'Race'.replace('R', 'Face')</code>
Letter check	<code>.istitle()</code>	Returns boolean value <i>True</i> , if the first character is a capital letter. Otherwise returns <i>False</i>	<code>'Martin'.istitle()</code> <i>True</i>
Letter change	<code>.swapcase()</code>	Swaps the upper and lowercase letters.	<code>'Mr. Jones PhD'.swapcase()</code> <i>'mR. jONES pHd.'</i>
Split by symbol	<code>.split('a')</code>	Splits the given string by the symbol given in the parentheses into a list. If the parentheses are empty, the method splits the list by space	<code>splitted = 'This is my car'.split()</code> <i>['This', 'is', 'my', 'car']</i>
Split by delimiter	<code>.splitlines([keepends])</code>	Splits the string on line end delimiters. Flag keepends - tells whether line delimiters ('\n' or '\r' etc.) should be kept	<code>'This is the first line\nThis is the second line'.splitlines(keepends = True)</code> <i>['This is the first line\n', 'This is the second line']</i>

48% z Lekce 1

Split by separator	<code>.partition(separator)</code>	<p>of part of the string preceding the separator, separator, part of the string succeeding the separator</p> <p>up-to- <code>date'.partition('-', 'to-date')</code></p>
Strip of symbols	<code>.strip()</code>	<p>Strips the given strings of the symbols we want to remove</p> <p><code>'www.example.cz'. 'example' .cz')</code></p>
Character type	<code>.isdecimal()</code>	<p>Returns True if all the characters inside the string are decimal characters <code>'1234'.isdecimal()</code> True ('0','1','2','3','4','5','6','7','8','9') and there is at least 1 character, otherwise False</p>
Character type	<code>.isdigit()</code>	<p>Returns True if all the characters inside the string are decimal characters or subscripts or superscripts (e.g. <code>"\u00B2"</code>) and there is at least one character, otherwise False.</p> <p><code>'\u00B2'.isdigit()</code> True</p>

48% z Lekce 1

Character type	<code>.isnumeric()</code>	<p>all the characters inside the string are decimal characters, subscripts, superscripts (e.g. <code>"\u00B2"</code>), fractions (e.g. <code>"\u00BC"</code>), roman numerals (e.g. <code>"\u2165"</code>), currency numerators etc. (all characters that have the Unicode numeric value property), and there is at least one character, False otherwise.</p>	<code>'\u2165'.isnumeric()</code> True
Character type	<code>.isalpha()</code>	<p>Returns boolean value <i>True</i>, if all the characters are letters. Otherwise returns <i>False</i></p>	<code>'asdxfb'.isaplha()</code> True

48% z Lekce 1

Character type	<code>.isalnum()</code>	<p>characters in the string are alphanumeric and there is at least one character, false otherwise</p> <p>(Basically all the Character Type methods above -</p> <p><code>.isdecimal()</code> , <code>.isdigit()</code> , <code>.isnumeric()</code> , <code>.isalpha()</code> .</p>	<code>'Z312'.isalnum()</code>	True
Search	<code>.find('m')</code>	Finds a character in a string and returns its index	<code>'Thomas'.find('m')</code>	3
Join by symbol	<code>.join(seq)</code>	<p>Joins the items of the sequence in the parentheses by the symbol written in the string</p>	<code>':' .join(['a', 'b'])</code>	'a:b'

48% z Lekce 1

Numeric data types

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [QUIZ](#) / [NUMERIC DATA TYPES](#)

1/11

What will be the result of the following expression?: 15/4

A. 3.75

B. 3.0

C. 3

D. 4

Booleans

48% z Lekce 1

1/16

Boolean data type is something like a subcategory of integer data type.

A. True

B. False

Sequences

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [QUIZ](#) / [SEQUENCES](#)

1/6

Strings...:

A. can be indexed, sliced and concatenated

B. are of the category mutable sequence

48% z Lekce 1

Concatenation

PYTHON ACADEMY / 1. INTRO TO PROGRAMMING / HOME EXERCISES - STRINGS / CONCATENATION

Create a python script named `fullname.py` that will perform the following actions:

- Store the string value **'Bob'** inside the variable `name` .
- Print the information that value **'Bob'** has been stored, for example: *"Storing 'Bob' in name ..."*
- Then store the string **'Marley'** a variable `surname` .
- Print the information that value **'Marley'** has been stored, for example: *"Storing 'Marley' in surname ..."*
- Lastly create a variable `full_name` , assign it the result of concatenation of values in variables `name` and `surname` and print the content of `full_name` to the terminal.

Example of script usage in terminal:

```
~/PythonBeginner/Lesson1$ python fullname.py
Storing 'Bob' in name ...
Storing 'Marley' in surname ...
```

48% z Lekce 1

Online Python Editor

You can create the script in you computer or here in our editor.

```
1  # Name
2
3
4  # Surname
5
6
7  # Full name|
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES](#) / [STRINGS](#) / [REpetition](#)

Create a python script named `print_header.py` that will perform the following actions:

Print the following header to the terminal. The result lines should contain 20 equal signs - both upper and the lower. The result should look something like:

```
=====
THIS IS THE HEADER
=====
```

You will probably want to use string repetition operation.

Example of script usage in terminal:

```
~/PythonBeginner/Lesson1 $ python print_header.py
=====
THIS IS THE HEADER
=====
```

Online Python Editor

You can create the script in you computer or here in our editor.

```
1 # Header|
```

48% z Lekce 1

[spustit kód](#)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

[Click to see our solution](#)

Indexing

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - STRINGS](#) / [INDEXING](#)

Create a python script named `get_day.py` that will perform the following actions.

Ask the user for the answer to the question:

What is the name of the fourth day in a week?

Of course, the correct answer in Europe is *Thursday*. Print `True` or `False` to the terminal based on whether the answer has been correct or not. Your script should print True if the user provided any of the following values: **Thursday, THURSDAY, thursday, tHursday, etc.**

Example of running the script in terminal:

```
~/PythonBeginner/Lesson1 $ python get_day.py
What is the name of the fourth day in a week? Thursday
```

48% z Lekce 1

```
~/PythonBeginner/Lesson1 $ python get_day.py  
What is the name of the fourth day in a week? Wednesday  
False
```

```
~/PythonBeginner/Lesson1 $ python get_day.py  
What is the name of the fourth day in a week? THURsday  
True
```

Online Python Editor

You can create the script in you computer or here in our editor.

```
1 # Days  
2  
3  
4 # Input|
```

spustit kód

48% z Lekce 1

[Click to see our solution](#)

Length

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - STRINGS](#) / [LENGTH](#)

Create a python script named `word_length.py` that will print out the length of the word 'quetzalcoatl' in a sentence similar to:

```
quetzalcoatl is X characters long
```

The placeholder X in the above string should be replaced by the actual length of the word.

Example of running the script in terminal:

```
~/PythonBeginner/Lesson1$ python word_length.py  
quetzalcoatl is 12 characters long
```

Online Python Editor

You can create the script in you computer or here in our editor.

```
1 # Get lenght  
2  
3  
4 # Print sentence|
```

48% z Lekce 1

[spustit kód](#)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

[Click to see our solution](#)

Slicing

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - STRINGS](#) / [SLICING](#)

Create a python script named `str_slicing.py` that will use indexing to:

- Extract the first five letters from the word 'tutorial'
- Print them to the terminal
- Extract the last five letters from the word 'approximation'
- Print them to the terminal
- Extract every third letter from the word 'approximation'

48% z Lekce 1

```
~/PythonBeginner/Lesson1$ python str_slicing.py  
First five letters: tutor  
Last five letters: ation  
Every 3rd letter: aritn
```

Online Python Editor

You can create the script in you computer or here in our editor.

```
1 # First five 'tutorial'  
2  
3  
4 # Last five 'approximation'  
5  
6  
7 # Every 3rd 'approximation'|
```

spustit kód

Code Solution

48% z Lekce 1

[Click to see our solution](#)

Letter Case

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - STRINGS](#) / [LETTER CASE](#)

Create a python script named `change_case.py` that will perform the following actions:

- ask the user for input
- transform all letters in the input into lowercase
- print the resulting lowercase string

Example of running the script in terminal:

```
~/PythonBeginner/Lesson1 $ python change_case.py
Enter something: Hello World
Changed to lowercase: hello world
```

Online Python Editor

You can create the script in you computer or here in our editor.

```
1 # Input
2
3
4 # Print|
```

48% z Lekce 1

[spustit kód](#)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

[Click to see our solution](#)

Searching

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - STRINGS](#) / [SEARCHING](#)

Create a python script named `find_t.py` that will perform the following actions:

- print out at what index is the first occurrence of the letter 't' in the word 'quetzalcoatl'
- print out at what index is the last occurrence of the letter 't' in the word 'quetzalcoatl'

Example of running the script in terminal:

```
~/PythonBeginner/Lesson1 $ python find_t.py
The first occurrence of the letter "t" is at the index: 3
The last occurrence of the letter "t" is at index: 10
```


48% z Lekce 1

```
1 # Word
2 word = 'quetzalcoatl'
3
4 # First occurrence
5
6
7 # Last occurrence
8
9
10 # Print|
```

[spustit kód](#)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

[Click to see our solution](#)

48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES](#) / [STRINGS](#) / [STRINGING](#)

Create a python script named `clean_word.py` that will remove any of the below listed characters from the end and the beginning of the string `'-hello-world. ,'`

- `' , '`
- `' . '`
- `' _ '`
- `' ' '`

Example of running the script in terminal:

```
/Users/PythonBeginner/Lesson1$ python clean_word.py
Original string: -hello-world. ,
Cleaned string: hello-world
```

Online Python Editor

You can create the script in you computer or here in our editor.

```
1 # Word
2 word = '-hello-world. ,'
3
4 # Cleaning
5
6
7 # Print|
```

48% z Lekce 1

[spustit kod](#)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

[Click to see our solution](#)

Replacing

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - STRINGS](#) / [REPLACING](#)

Create a python script named `replace_name.py` that will store a string in a variable:

```
'Hello Bob! Name Bob is my preferred name.'
```

- Ask for the user to enter his/her name
- Replace all occurrences of string 'Bob' with the name provided by the user

Example of running the script in terminal:

```
~/PythonBeginner/Lesson1 $ python replace_name.py
What is your name? Frank
Hello Frank! Name Frank is my preferred name.
```

Online Python Editor

You can create the script in you computer or here in our editor.

48% z Lekce 1

```
4  # Input
5
6
7  # Replacement
8
9
10 # Print|
```

[spustit kód](#)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Creating lists

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - LISTS](#) / [CREATING LISTS](#)

For the following exercises it will be better if you do them locally in your computer. Open a new file in your code editor and name it `employees.py`. The following tasks should be written inside this single file.

Let's start.

- Assign an empty list to the variable `candidates`
- Print the content of `candidates` to the terminal introducing it with the string: `Candidates at the beginning:`
- Create a list containing strings: 'Frank', 'Amy', 'John', 'Kate' and assign this list to the variable `employees`
- Print the content of `employees` to the terminal introducing it with the string: `Employees at the beginning:`

Example of running the script:

48% z Lekce 1

```
employees at the beginning: [ Frank , Amy , John , Kate ]
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Adding items to lists

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - LISTS](#) / [ADDING ITEMS TO LISTS](#)

Please, continue working with the file `employees.py`. Now, write a code that will perform the following actions:

- add new names 'Bob' and 'Ann' into the list `candidates`
- print the content of the variable `candidates` introduced by the string: `Added new names to candidates:`

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python employees.py
Candidates at the beginning: []
Employees at the beginning: ['Frank', 'Amy', 'John', 'Kate']
Added new names to candidates: ['Bob', 'Ann']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Inserting items to lists

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - LISTS](#) / [INSERTING ITEMS TO LISTS](#)

Please, continue working with the file `employees.py`. Now, write a code that will perform the following actions:

- insert name 'Bob' stored in the variable `candidates` into the `employees` list at the index 1
- print the content of the variable `employees` introduced by the string: `Added new names to employees:`

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python employees.py
Candidates at the beginning: []
Employees at the beginning: ['Frank', 'Amy', 'John', 'Kate']
Added new names to candidates: ['Bob', 'Ann']
Added new names to employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Removing items from lists

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - LISTS](#) / [REMOVING ITEMS FROM LISTS](#)

Please, continue working with the file `employees.py`. Now, write a code that will perform the following actions:

48% z Lekce 1

candidates:

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python employees.py
Candidates at the beginning: []
Employees at the beginning: ['Frank', 'Amy', 'John', 'Kate']
Added new names to candidates: ['Bob', 'Ann']
Added new names to employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate']
Removed Bob vfrom candidates: ['Ann']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Repeating items in list

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - LISTS](#) / [REPEATING ITEMS IN LIST](#)

Please, continue working with the file **employees.py**. Now, write a code that will perform the following actions:

- repeat the name 'Ann' 3 times inside the list **candidates**
- print the content of the variable **candidates** introduced by the string: **Repeated name Ann in candidates:**

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python employees.py
Candidates at the beginning: []
```


48% z Lekce 1

```
Added new names to employees: [ Frank , Bob , Amy , John , Kate ]  
Removed name from candidates: ['Ann']  
Repeated name Ann in candidates: ['Ann', 'Ann', 'Ann']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Concatenating lists

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - LISTS](#) / [CONCATENATING LISTS](#)

Please, continue working with the file `employees.py`. Now, write a code that will perform the following actions:

- merge the list `candidates` into the list `employees`
- print the content of the variable `employees` introduced by the string: `Merged candidates with employees:`

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python employees.py  
Candidates at the beginning: []  
Employees at the beginning: ['Frank', 'Amy', 'John', 'Kate']  
Added new names to candidates: ['Bob', 'Ann']  
Added new names to employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate']  
Removed name from candidates: ['Ann']  
Repeated name Ann in candidates: ['Ann', 'Ann', 'Ann']  
Merged candidates with employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate',  
'Ann', 'Ann', 'Ann']
```

48% z Lekce 1

[Click to see our solution](#)

Indexing with lists

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - LISTS](#) / [INDEXING WITH LISTS](#)

Please, continue working with the file `employees.py`. Now, write a code that will perform the following actions:

- print out the name at the index 2 introduced by the string: **At index 2 we have:**
- print out the name at the last index introduced by the string: **At index <index_num> we have:**

The placeholder `<index_num>` should be replaced by the index number of the last position in the list `employees`

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python employees.py
Candidates at the beginning: []
Employees at the beginning: ['Frank', 'Amy', 'John', 'Kate']
Added new names to candidates: ['Bob', 'Ann']
Added new names to employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate']
Removed name from candidates: ['Ann']
Repeated name Ann in candidates: ['Ann', 'Ann', 'Ann']
Merged candidates with employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate',
'Ann', 'Ann', 'Ann']
At index 2 we have: Amy
At index 7 we have: Ann
```

48% z Lekce 1

[Click to see our solution](#)

Slicing lists

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - LISTS](#) / [SLICING LISTS](#)

Please, continue working with the file `employees.py`. Now, write a code that will perform the following actions:

- print out the names at the indices 2 to 5 introduced by the string: **At indices 2 to 5 we have:**

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python employees.py
Candidates at the beginning: []
Employees at the beginning: ['Frank', 'Amy', 'John', 'Kate']
Added new names to candidates: ['Bob', 'Ann']
Added new names to employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate']
Removed name from candidates: ['Ann']
Repeated name Ann in candidates: ['Ann', 'Ann', 'Ann']
Merged candidates with employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate',
'Ann', 'Ann', 'Ann']
At index 2 we have: Amy
At index 7 we have: Ann
At indices 2 to 5 we have: ['Amy', 'John', 'Kate', 'Ann']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Striding lists

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - LISTS](#) / [STRIDING LISTS](#)

Please, continue working with the file `employees.py`. Now, write a code that will perform the following actions:

- print out every third string in `employees` introduced by the string: **Every third member:**

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python employees.py
Candidates at the beginning: []
Employees at the beginning: ['Frank', 'Amy', 'John', 'Kate']
Added new names to candidates: ['Bob', 'Ann']
Added new names to employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate']
Removed name from candidates: ['Ann']
Repeated name Ann in candidates: ['Ann', 'Ann', 'Ann']
Merged candidates with employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate',
'Ann', 'Ann', 'Ann']
At index 2 we have: Amy
At index 7 we have: Ann
At indices 2 to 5 we have: ['Amy', 'John', 'Kate', 'Ann']
Every third member: ['Frank', 'John', 'Ann']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



48% z Lekce 1

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES](#) / [LISTS](#) / [INDEX DETERMINATION](#)

Please, continue working with the file `employees.py`. Now, write a code that will perform the following actions:

- print the index, at which the string 'John' encounters itself in `employees`: **John is at index:**

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python employees.py
Candidates at the beginning: []
Employees at the beginning: ['Frank', 'Amy', 'John', 'Kate']
Added new names to candidates: ['Bob', 'Ann']
Added new names to employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate']
Removed name from candidates: ['Ann']
Repeated name Ann in candidates: ['Ann', 'Ann', 'Ann']
Merged candidates with employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate',
'Ann', 'Ann', 'Ann']
At index 2 we have: Amy
At index 7 we have: Ann
At indices 2 to 5 we have: ['Amy', 'John', 'Kate', 'Ann']
Every third member: ['Frank', 'John', 'Ann']
John is at index: 3
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Counting items

48% z Lekce 1

Please, continue working with the file `employees.py`. Now, write a code that will perform the following actions:

- print how many times name 'Ann' can be found in `employees`: **The number of occurrences of Ann:**

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python employees.py
Candidates at the beginning: []
Employees at the beginning: ['Frank', 'Amy', 'John', 'Kate']
Added new names to candidates: ['Bob', 'Ann']
Added new names to employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate']
Removed name from candidates: ['Ann']
Repeated name Ann in candidates: ['Ann', 'Ann', 'Ann']
Merged candidates with employees: ['Frank', 'Bob', 'Amy', 'John', 'Kate', 'Ann', 'Ann', 'Ann']
At index 2 we have: Amy
At index 7 we have: Ann
At indices 2 to 5 we have: ['Amy', 'John', 'Kate', 'Ann']
Every third member: ['Frank', 'John', 'Ann']
John is at index: 3
The number of occurrences of Ann: 3
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Complete Solution

[PYTHON ACADEMY](#) / [1. INTRO TO PROGRAMMING](#) / [HOME EXERCISES - LISTS](#) / [COMPLETE SOLUTION](#)

48% z Lekce 1

Creating list



Adding



Inserting



Removing



Repeating



Concatenating



Indexing



Slicing



Striding



48% z Lekce 1

Counting



ENTIRE CODE



DALŠÍ LEKCE