Osnova

Lesson Overview

PYTHON ACADEMY / 10. IMPORTING / LESSON OVERVIEW

Welcome to lesson 10. Major theme of this lesson is modules. Modules are blocks of code typed by someone else. You can edit these blocks, add some functionality and so on. But first you need to **import** modules.

You will learn:

- how to **import modules**,
- which **types of modules are the most used**,
- how to **create your own module**.

100% z Lekce 12

Osnova

REVIEW EXERCISES

Errorless Int Converter

PYTHON ACADEMY / 10. IMPORTING / REVIEW EXERCISES / ERRORLESS INT CONVERTER

Create a function that accepts a list of strings and returns only those that are convertible to integer. Use try statement to solve this task. Also the input list can have any number of strings.

Example of function in use:

```
>>> int_convert('Hello', '23','12', 'Bob', 'new')
```

100% z Lekce 12

Osnova

Online Python Editor

spustit kód

Code Solution

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

Click to see our solution

100% z Lekce 12

Osnova

```
          vert(*items):
          = []
    for item in items:
        try:
            result.append(int(item))
        except ValueError:
            pass
    return result
```

Don't forget that in order your script works, you need to unpack the list you are sending into the funcion. If you're unsure how unpacking works check out the unit about unpacking inputs.

Simulating Syntax Checker

PYTHON ACADEMY / 10. IMPORTING / REVIEW EXERCISES / SIMULATING SYNTAX CHECKER

In this task, we will put ourselves in shoes of simple **syntax checker developer**. Our task is to create a function, that is given a string representing an expression containing:

- brackets/parentheses
- values
- operators

The function should evaluate, whether opening parentheses are correctly matched with their closing counterparts.

Example of running the function:

```
>>> expr = "5 + (a - b) * ((64-2) + 5)"
>>> check(expr)
True
>>> expr = "5 + (a - b) * ((64-2 + 5)"
>>> check(expr)
```

100% z Lekce 12

C - f' ' ' ulc' xpect that there is only one type of opening and closing brackets in the Osnova ,re efc the main probem will be with insufficient number of opening or closing brackets.

You can test your function using the following **test cases**:

```
{'abc', 'efg', {1,2}, 'ij'},32} --> False
(54, 65 + 96, 'hello'), 'world') --> False
((('a','b')),'c'))), 'e') --> False
[[[['a,'b'],'c']]] --> True
```

Bonus

Make sure that your function is able to **distinguish** among different kinds of brackets - { }, [ ], ( )

Example of matching opening and closing brackets - function should return `True` :

```
>>> expr = "{'name':'John', 'address' : {'street':'Main',
'City':'London','number':123}, 'performance': [3,1,2,1,1]}"
>>> check(expr)
True
```

However, the function should return `False` , if outer closing bracket is inserted earlier than the inner closing bracket:

```
>>> expr = "{'name':'John', 'address' : {'street':'Main',
'City':'London','number':123}, 'performance': [3,1,2,1,1}]"
>>> check(expr)
False
```

You can test your function using the following **test cases**:

```
{'abc', 'efg',[1,2]], 'ij'),32} --> False
[['a',{'b'}},'c'] --> False
[[[{['a'],'b'},'c']],'d'] --> True
[[{[['a'],'b'},'c']],'d'] --> False
```

100% z Lekce 12

```
1
```

Osnova

spustit kód

Code Solution

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

Click to see our solution

```
def check(expr):
    brackets = {'(':')', '{':'}','[':']'}
    stack = ['']
```

100% z Lekce 12

Osnova

```
            har in brackets:
                stack.append(brackets[char])

            elif char in brackets.values() and char != stack.pop():
                return False

    return stack == ['']
```

ONSITE EXERCISES

Our Goal

100% z Lekce 12

We would like to create two programs today:

Osnova

1. Program that will search for a directory or a file with a given name
2. Program that will copy a list of files or directories into a given folder

These two programs can regarded as complementary. The first one will collect all the target files and folders, the other one will copy them into a given destination.

First we will have to create few files and directories to be used later on when testing our programs.

Getting to the needed functions

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / GETTING TO THE NEEDED FUNCTIONS

We want to create and move our files around. To do that, we need to learn, how to access Python functions that will allow us to do those things.

We need to **import** those functions. We already know, how to get to the functions that generate random numbers:

```
import random
```

We have imported so called **module**. Module, which name is `random` . First we should speak about, what a module actually is and how does it look like.

Modules are Python files. Modules, that are installed along with Python create Python Standard Library. If we wanted to see, how such files look like, we can look into the folder C:\Program Files\Python\Lib on Windows or /usr/lib/python_version on Ubuntu. This folder contains those modules we are interested in.

Today we will work with three modules:

- **os** and **os.path** - working with paths, navigating accros directories, creating directories, renaming files etc.

100% z Lekce 12

ʼinˑ ˑˡes and directories

Osnova

Therefore we can import these as follows:

```
import os
import sys
import shutil
```

How Python Gets the Modules for us

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / HOW PYTHON GETS THE MODULES FOR US

In order we can begin to use the functionality, that comes with imported module Python has to perform few steps:

1. It has to find a file or directory with the name of the imported module
2. It has to load the corresponding file into our program

Search

Once the Python program is turned on, Python already loads some functionality and data automatically along the way. Among that data, there is

- information about the modules, that have already been imported into the program
- information about where to look for modules to be imported

Information mentioned in the first point is available in the variable `sys.modules`

```
sys.modules
```

Information locating the places, where will Python search for modules to be imported can be encountered in `sys.path` :

```
sys.path
```

100% z Lekce 12

Osnova ctories

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / CREATING DIRECTORIES

To create directories we will need to import module `os` :

```
import os
```

To create a directory we will use the command `os.mkdir` :

```
os.mkdir(path)
```

The variable `path` should be a string representing the path at which a new directory is to be created. Keep in mind, that the path needs to mbe closed in quotation marks, eg. `os.mkdir('/home/PythonBeginner/DIR')`

- What if the directory already exists?
- What if we supply a path string, where some of the directories on the path do not exist?

Task

- Create a directory called **DIR** inside the directory **PythonBeginner/Lesson8**
- Create 3 directories called **DIR1**, **DIR2**, **DIR3** inside the directory **DIR**.
- Indiside the directory **DIR1** create another directory called **DIR1.1**.

Creating Files

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / CREATING FILES

Function `open()` with **mode** parameter set to `'w'` or `'a'` will be sufficient for creating a new file.

100% z Lekce 12

C ate th f ll ing fi es:
Osnova

- in **DIR1**: test1.txt, readme.md
- in **DIR1.1**: test1.1.txt, app.config
- in **DIR2**: test2.txt, about.txt
- in **DIR3**: test3.txt, app.log

Visiting All Items in a Directory

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / VISITING ALL ITEMS IN A DIRECTORY

One of the corner stones of our programs will be their ability to traverse the file system - go from one directory into another.

To be able to do that, we need to have a listing of a directory's contents. This is where we use the function `os.listdir()` .

- To **list the contents** of the directory **DIR1** we can use the following command:

```
os.listdir('DIR1')
```

Of course, a directory of that name has to be present in the current working directory.

- And how do we find out, what is our **current working directory**? Using `os.getcwd()` :

```
os.getcwd()
```

- If you are not satisfied with your CWD, you can **change** it using `os.chdir()` . Below we move into the directory **DIR1/DIR1.1**:

```
os.chdir('DIR1/DIR1.1')
```

TASK

100% z Lekce 12

[illegible] to [illegible]ate over the contents of the directory **DIR2**, printing each of its items on a [illegible]

Osnova

Is it a File or a Folder?

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / IS IT A FILE OR A FOLDER?

It is not possible to use the function `os.listdir()` on a file. If we do that, we get `NotADirectoryError`. We can solve this in two ways, using `try-except` statement or a condition. We will use condition here, because we want our future program to record the information whether an item is a directory or a file.

To find out, whether an item represented by a path string is a file or a directory, we use the following two functions from `os.path` module:

Check for file (returns True)

```
os.path.isfile(path)
```

Check for directory (returns True)

```
os.path.isdir(path)
```

If the given path does not exist, both functions return `False`.

But what if it does not exist?

One of the inputs into our future programs will be the name of the folder from which the search should begin or the name of the folder, into which we want to copy files and folders. But what if the given folder path is incorrect and it does not exist? Then none of those programs will work. **We need to be able to check, whether a given path exists**:

```
os.path.exists(path)
```

100% z Lekce 12

What if ... got relative path?

Osnova

Another problem can arise, if our program is fed with a relative path. Relative path is always interpreted in regards to the current working directory of the running program. But what if the user did not count with that and the given relative path will not lead us to the destination? **We need to check, whether a given path is absolute**:

```
os.path.isabs(path)
```

TASK

1. Create a function that will print the contents of all the directories in the directory **DIR**. Remember, that only folder can be listed, not a file.
2. Improve the function in the way, that it will check, whether the directory path, from which the iteration starts, is absolute.

Creating Paths

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / CREATING PATHS

In the previous section, we were not able to list contents of the directories, that were not present in the current working directory. Therefore, we need a function, that will help us to create an absolute path to that external location.

To create a new path from given strings, we use function `os.path.join()`. Below, we are creating absolute path that will lead us into the folder **DIR1**. The prerequisite is that our current working directory is **DIR**:

```
os.path.join(os.getcwd(),'DIR1')
```

TASK

Finish the function from the previous section

100% z Lekce 12

1. C   ncti   that will print the contents of all the directories in the directory **DIR**.
   Osnova
   th   nly folder can be listed, not a file.

2. Improve the function in the way, that it will check, whether the directory path, from which the iteration starts, is absolute.

Copying Files & Folders

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / COPYING FILES & FOLDERS

To copy both files and folders, we will need to import new module - `shutil` :

```
import shutil
```

Copy file

```
shutil.copy(source,destination)
```

Copy Directory

```
shutil.copytree(source,destination)
```

TASK

1. Our function that iterates over all the directories under a given root directory should return a list of paths to all the files and folders from the DIR directory that contain string `'test'` in their name.

2. The list returned by the above function should be fed into another function that will:

   - create a new directory called BACKUP and
   - will copy all the listed items into the directory BACKUP.

100% z Lekce 12

Osnova `es & Folders

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / REMOVING FILES & FOLDERS

If a folder we want to create already exists, we get an `FileExistsError`. We may want to handle that error, and ask the user to decide, whether the folder should be removed and recreated again.

To remove a folder, we should use `shutil.rmtree()` function.

```
shutil.rmtree(path)
```

TASK

Let's implement into our copying function the possibility to recreate a folder if it already exists.

Importing our newly created functions

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / IMPORTING OUR NEWLY CREATED FUNCTIONS

So far we have always imported the whole module and accessed its functions through the dot following the module's name. But what if we wanted to import only a specific function. We would use the following syntax:

```
from module import function_name
```

We can try it with our newly created functions. Let's try to import one of them into running Python shell.

__name__ == '__main__'

100% z Lekce 12

W [illegible] pr [illegible] nt [illegible] avoid the execution of calls to functions, we are currently importing. In [illegible] Osnova [illegible] ity to put calls to those functions into a condition, that will evaluate true only if the python file is ran directly as the principal program and not as a Python file. The conditions looks like this:

```python
if __name__ == '__main__':
    here go our calls and other statements not to be executed during the import
```

Summary Importing

PYTHON ACADEMY / 10. IMPORTING / ONSITE EXERCISES / SUMMARY IMPORTING

During this lesson we will create our own Python module. Every module should have a common purpose. There are modules dedicated to multi-process programming, parsing different data formats (HTML, CSV, XML, JSON etc.), sending HTTP requests, testing and much more.

Our goal today will be to create a small module that will work with paths. The module should provide the following functionality:

1. Create absolute or relative path from given folder and file names
2. Determine, whether a given path is absolute or relative
3. Extract file name from a path
4. Extract file extension from a path
5. Extract nth item from a path

Absolute path

Represents a location of a file or a folder in a file system relative to the root directory.

Root directory

100% z Lekce 12

- with drive name as root - `C:`

Osnova

```
C:\User\Documents\report.xls
```

- Linux / Mac - begins with forward slash `/` as root directory

```
/usr/bin/env
```

Relative path

Represents a location of a file or a folder in a file system relative to the current working directory (CWD).

Current Working Directory

Directory in which the program being executed resides.

Example of relative path:

- does not begin with root directory

```
TestDir1/file.txt
```

Use of sys.platform

In our module, we will need to import standard library module called `sys`. We will use the variable `platform` provided by `sys` module to determine, what operating system is our script running on.

These are the values stored in `sys.platform` depending on what OS do we call it:

System	platform value
Linux	'linux'
Windows	'win32'
Mac OS X	'darwin'

100% z Lekce 12

Osnova

Character used to separate folders and files in a path are backslash for Windows " and forward slash for Linux / Mac `/` . The information about path separator relevant to the underlying OS is stores inside `os.sep` variable - in `os` module.

Create our module

Let's call our module `pypath.py` and store it in the directory `Lesson8` .

Then we will download the file `test_pypath.py` into `Lesson8` as well. This file contains calls to all the functions that our `pypath.py` should implement.

Importing a whole module

We can load all the variables and functions from an external module using the **`import` statement**:

```
import modulename
```

In our case we would import `pypath.py` into `test_pypath.py` :

```
import pypath
```

At the moment we are not able to use any

Create absolute path / relative path

Create a function `join_path()` that will accept

- variable number of positional arguments in form of strings representing folder and finally file name
- boolean argument, that will tell the function, whether absolute or relative path is to be created - default parameter value is `True`
- platform name ('win', 'linux', 'darwin') - default parameter is `sys.platform`

100% z Lekce 12

```
 >>          h(*['lib', 'lsb', 'init-functions'])
 Osnova
 /l          it   nctions'
>>> join_path(*['lib', 'lsb', 'init-functions'])
'lib/lsb/init-functions'
```

The above examples were run on Linux, therefore the root directory is `/` .

Testing your module in Python console

Sometimes it is faster to test changes made to our code in a console. We can import `pypath.py` into the console

```
>>> import pypath
>>> pypath.join_path(*['lib', 'lsb', 'init-functions'])
'/lib/lsb/init-functions'
```

To perform the above import, we our current working directory should be set to `Lesson8` . If we change anything in our `pypath` module, we can reload this module using `importlib.reload()` function:

```
>>> import importlib as imlib
>>> imlib.reload(pypath)
<module 'pypath' from '/home/martin/PythonBeginner/Lesson8/pypath.py'>
```

In the above import we have used aliasing - we have given a long module name a shorter one. This name only exists in currently running Python console.

Importing only selected attributes

To import only selected functions or variables from a module we use the following import statement:

`from modulename import attribute`

or

`from modulename import attr1, attr2, etc.`

100% z Lekce 12

function `isabs()` that will accept two arguments -

Osnova

- string representing path. The function should tell us, whether the given path is absolute or relative.
- platform name ('win', 'linux', 'darwin') - default parameter is `sys.platform`

Let's import the `isabs` function only into our console:

```
>>> from pypath import isabs
```

And let's check, whether it works properly.

And now we check the function using data from `test_pypath.py` :

```
>>> isabs(r'C:\Users\user\Pictures\123_ae.jpg',platform='win')
True
>>> isabs(r'Users\user\Pictures\123_ae.jpg',platform='win')
False
```

The `reload` function works only for modules, not attributes. Therefore we could not use it here.

Importing all attributes

We import all module attributes without having to list them all inside the import statement as follows:

```
from modulename import *
```

This import is highly discouraged to use, because it can lead to namespace clashes as well as confusion among script readers. Readers will not know, from which module, which functions have been imported.

Despite that let's perform this import inside `test_pypath.py` file and we will now continue working here.

```
from pypath import *
```

100% z Lekce 12

Extr[illegible] fi[illegible]name

Osnova

Function `basename()` should take the following arguments:

- string representing path
- path seprator - default value should be the separator corresponding to underlying OS

Example:

```
>>> from pypath import basename
>>> basename('/home/music/album3/song2.mp3')
'song2.mp3'
```

Extract the extension

The function `splitext()` accepts only one argument - path to be split - and returns a tuple in form (base, file extension). Base in this case is everything that precedes the file extension. If there is not file extension, the base should be the original string and the second item should be an empty string:

```
>>> splitext(r'C:\Program Files\Lenovo\OneKey\recover.exe')
('C:\\Program Files\\Lenovo\\OneKey\\recover', 'exe')
>>> splitext('/bin/mkdir')
('/bin/mkdir', '')
```

Extract the nth item in a path

Finally, we would like to implement the functionality, that returns the nth item in a path. The function should accept three inputs:

- path to be manipulated
- number refering to nth item in the path
- path separator - default value should be the separator for underlying OS

Example:

100% z Lekce 12

```
>>> extract nth('/home/music/album1/song2.mp3',4)
```

Osnova

```
>>> extract_nth(r'C:\Program Files\Lenovo\OneKey\recover.exe',2,sep='\\')
'Program Files'
```

Main script

To make sure, that some code is not executed if a file is imported, we use the `if __name__ == '__main__'` condition. Everything that is inside its block is executed only if the file is run directly as a script and not imported.

We can check this inside the `pypath.py` module. We can transfer there all the testing code from `test_pypath.py` and indent it in order it is considered to pertain under `if __name__ == '__main__'` . Now try to import the `pypath` into newly open console.

IMPORTING MODULES

100% z Lekce 12

Osnova dule What is a Module

Module is any Python file - any file with .py suffix. It normally contains variables, function definitions etc. Sometimes, we want to use those functions or variables in another file. However, we do not want to copy and paste the same code around all our scripts. We just want to load them from that single python file into our running program.

And this is where importing comes to play. Loading objects from one file into another is called **importing**. It is performed using the `import` statement and the name of the file we want to import :

```
import python_module
```

For example, we have already used `import` statement to load functionality from the module called `random` :

```
>>> import random
>>> random.randint(1,10)
2
```

Module Advantages

You might ask, why modules? Similarly to functions, modules allow us to group logically related functionality together. Compared to functions, they are one level higher in content organization. One module can contain multiple functions, or variables and other objects. So the advantages are then:

- **Code organization** - related functionality is kept in one file - e.g. functionality that generates random values inside the `random` module
- **Code reusability** - the code inside one module can be imported across multiple scripts

100% z Lekce 12

Standard vs 3rd Party Modules

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES / STANDARD VS 3RD PARTY MODULES

In Python, we very often import functionality from so called libraries. These can be:

- **standard Python library**, that comes with the Python installation (f.e. `sys` module),
- or **third party library** module (f.e. `requests` module) which are not part of the standard library, but provide useful and many times additional functionality to the standard library

Where can we find all those modules?

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES / WHERE CAN WE FIND ALL THOSE MODULES?

As every operating system installs Python directory somewhere else, the best way, how to find out, where the Python modules reside is the invoke the `path` variable from the `sys` module:

```
>>> import sys
>>> sys.path
```

In windows, Python folder can be usually found under `C:\Program Files\Python`. Once we are inside the folder **Python**, we should see the **Lib** directory, that contains all the Python Standard Library modules as well as special directory called **site-packages**, where Python installs modules, that do not come with Python Standard Library.

Import Process

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES / IMPORT PROCESS

When Python imports module into another module, it performs these two actions:

1. Search for the imported module
2. Module compilation into bytecode

3. Running the code found inside the imported module

All this loading steps take place only the first time when import statement is encountered during the execution of a program. If the same import statement is encountered later during the session, Python just fetches module object already available in the memory from `sys.modules` variable.

Module Search

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES / MODULE SEARCH

Python performs the following 3 steps in search for imported module:

1. look into `sys.modules`
2. if not found, look into `sys.path`
3. if not found, raise `ImportError`

sys.modules

In order to save time and energy, Python first looks for the imported module in the variable calles `sys.modules`. The `sys.modules` variable is a dictionary containing all the modules that have been imported since Python was started. The key is the module name, the value is the module object.

If we want to see the content of this variable, we have to import the library called `sys` at first:

```
>>> import sys
>>> sys.modules
{'builtins': <module 'builtins' (built-in)>, 'sys': <module 'sys' (built-in)>, '_frozen_importlib': <module '_frozen_importlib' (frozen)>, '_imp': <module '_imp' (built-in)>, '_warnings': <module '_warnings' (built-in)>, '_thread': <module '_thread' (built-in)>, ....
```

100% z Lekce 12

If not found in sys.modules , Python then searches places listed in the sys.path variable. This variable contains list of paths to folders that contain Python code. On Linux Ubuntu with Anaconda distribution of Python, these are the folders:

```
>>> sys.path
['', '/home/martin/anaconda3/lib/python35.zip',
'/home/martin/anaconda3/lib/python3.5',
'/home/martin/anaconda3/lib/python3.5/plat-linux',
'/home/martin/anaconda3/lib/python3.5/lib-dynload',
'/home/martin/anaconda3/lib/python3.5/site-packages',
'/home/martin/anaconda3/lib/python3.5/site-packages/PyDispatcher-2.0.5-
py3.5.egg', '/home/martin/anaconda3/lib/python3.5/site-packages/Sphinx-
1.3.5-py3.5.egg', '/home/martin/anaconda3/lib/python3.5/site-
packages/setuptools-20.3-py3.5.egg']
```

You may note that the first item in the list is an empty string. Python understands that as: "Look into the home directory". Home directory is in this case place, where the main Python file of currently running program resides.

For example a script called `mortgage.py` is located in a folder called `app` . The folder looks like this:

```
app/
  |___mortgage.py
  |___random.py
```

We launch the `mortgage.py` file from the terminal:

```
~/app $ python mortgage.py
```

The file `mortgage.py` contains command `import random` . We already know, that Python standard library contains module called `random` . If we however, create our own `random.py` file in the same folder as the running `mortgage.py` , Python will first search in the **home directory** of `mortgage.py` , where it will find a module of such name. Therefore, we will not get the functionality of standard library module, that resides in `Python\Lib` folder, but the functionality contained inside the `app\random.py` .

100% z Lekce 12

If the module is not found neither in any of the listed directories, `ImportError` is raised.

If the module is not found neither in any of the listed directories, ImportError is raised.

Module Loading

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES / MODULE LOADING

Once the module is found, it has to be loaded into the importing program. The goal of loading the module into a program is a creation of module object.

Python creates a module object inside the executing program's namespace and associates it with the imported module name. Once we have the module object stored inside the variable, we can work with it as with any other Python variable / object.

In the below sections, we will be working with Python file called `test.py` (stored inside the folder `/PythonBeginner/Lesson8`), which contains the following lines of code:

```python
print('Print is being executed')
var1 = 52
var2 = 53

def func(v1,v2):
    result = v1 + v2
    return result
```

Now we launch interactive Python console in our terminal:

```
~/PythonBeginner/Lesson8 $ python
```

And we import a module called `test` (our goal is to import module `test.py` which resides in the current working directory - Lesson8):

```
>>> import test
```

What happens during the loading

What happens during the loading

100% z Lekce 12

Osnova

When Python imports the file it has to compile and actually **execute** the code inside it. Therefore importing the file `test.py` causes the message `'Print is being executed'` to be printed to the terminal:

When Python imports the file it has to compile and actually execute the code inside it. Therefore
importing the file test.py causes the message 'Print is being executed' to be printed
to the terminal:

```
>>> import test
Print is being executed
```

What is loaded?

Every variable declared in the global scope of the imported module is loaded into the importing script. These variables are then accessible as imported object attributes using the dot notation.

After we have run `import.py`, we should have access to all global variables from `test.py`. The global variables in this case are `var1`, `var2`, `func`:

```
>>> test.var1
52
>>> test.var2
53
>>> test.func
<function func at 0x7ff2afa6bbf8>
```

And we should not have access to local variable `result` from the function `func`:

```
>>> test.result
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'test' has no attribute 'result'
>>> test.func.result
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'function' object has no attribute 'result'
```

Module Object Attributes

Module Object Attributes

100% z Lekce 12

' the ect' we worked with the module `test.py` stored inside the current working
Osnova
ec ib nner/Lesson8) and we will use it here as well:

In the previous section we worked with the module test.py stored inside the current working directory (PythonBeginner/Lesson8) and we will use it here as well:

```python
var1 = 52
var2 = 53

def func(v1,v2):
    result = v1 + v2
    return result
```

We know, that we can access variables or functions in the imported module using the dot notation: `module.attribute` . Specifying the module name, then appending the attribute name.

However **to inspect the module object for all the attributes** it contains, we can use special attribute `__dict__` . This attributes contains the dictionary representing module's namespace - names of global variable names as dictionary keys and corresponding objects as values.

```
>>> import test
>>> test.__dict__
{'__name__': 'test', '__doc__': None, '__package__': '',  ....
'var1': 52, 'var2': 53, 'func': <function func at 0x7ff2afa6bbf8>}
```

The module's namespace contains also many objects, that we have not specified in its script, but we do not have to worry about that now. **The important part is that we can see our global variables listed at the end of the namespace dictionary.**

if __name__ == "__main__"

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES / IF __NAME__ == "__MAIN__"

The Python file can serve as importable module as well as an executable script. Let's say we have a script `test.py` , that not only declares variables and defines functions, but also calls those functions inside itself:

```python
var1 = 52
```

100% z Lekce 12

Osnova

```
def func(v1,v2):
    result = v1 * v2
    return result

print(func(var1,var2))
```

1 var1 = 52
2 var2 = 53
3
4 def func(v1,v2):
5 result = v1 * v2
6 return result
7
8 print(func(var1,var2))

Importing the above module ends up as follows:

```
>>> import test
2756
```

If we just wanted to import this script as a module into another file, we would probably like to avoid the call `func(var1,var2)` to be executed during the import process.

We can avoid the statement `func(var1,var2)` to be run during the import if we enclose it into the following condition:

```
if __name__ == "__main__":
    print(func(var1,var2))
```

We can still run the file as a script and we still get printed the result:

```
~/PythonBeginner/Lesson8 $ python test.py
2756
```

But when imported, the print statement is not executed:

```
>>> import test
>>> test.var1
52
```

Those statements we do not want to be executed during the import, have to be placed inside the condition `if __name__ == "__main__":` placed at the bottom of our Python script. Such Python files can serve as executable scripts as well as importable modules.

Content of __name__

Content of __name__

100% z Lekce 12

Every module has access to its own variable __name__ . This variable tells us, whether a given module is executed as a script (from a comand line) or it is imported as a module.

We will demonstrate the case on the module `mod.py` , which contains the following line of code:

```python
print('Execution of the file: ' + __name__)
```

Now we will run this file **as a script** from the terminal:

```
~/PythonBeginner/Lesson8 $ python mod.py
```

And this gets printed to the terminal:

```
Execution of the file: __main__
```

And now we will **import the file** into Python interactive console:

```
>>> import mod
Execution of the file: mod
>>>
```

We can see that:

- if the file is run as a script (main program), the value of variable `__name__` for this module is `__main__` .
- if we import the file, everything that is found inside it is executed as well, but this time, the variable `__name__` refers to the module name - in our case `mod`

Module Names

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES / MODULE NAMES

Python module names must follow the variable naming rules in Python. This is because **the file name becomes variable name inside the program** that imports it.

100% z Lekce 12

- re ved keyword,

Osnova

- a name that would begin with a number,
- or a name that would contain a space character.

a name of a reserved keyword,
a name that would begin with a number,
or a name that would contain a space character.

It is also good to name your Python files differently than the standard library modules or 3rd party modules, in order to avoid name clashes during the import.

Import Statements

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES / IMPORT STATEMENTS

There a multiple ways how external code can be imported into a program:

1. Importing modules
2. Importing module attributes only
3. Importing modules or attributes under alias

All import statements should be listed at the top of our Python scripts:

```
import sys
from random import randint

...
your code
...
```

Importing Modules

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES / IMPORTING MODULES

We can import single or multiple modules.

We can import single or multiple modules.

100% z Lekce 12

When we want to import single module Python searches for a file called modulename, creates the module object and assigns it to a variable modulename. It looks as follows:

```python
import modulename
```

If we wanted to import **more than one module** into our program, we can:

- do it all in one import statement - module names separated by comma:

```python
import modulename1, modulename2, ...
```

- import each module on a separate line:

```python
import modulename1
import modulename2
import ...
```

From the python style point of view, this is not recommended to list many names in one import statement as it is less obvious what modules are actually imported.

In all the above cases we can access the module attribute through `module.attribute` notation.

Importing Module Attributes

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES / IMPORTING MODULE ATTRIBUTES

Demonstration in this section will be performed on a file `test.py` :

```python
print('Print is being executed')
var1 = 52
var2 = 53

def func(v1,v2):
    result = v1 + v2
```

100% z Lekce 12

To import module attributes (module globall variables), we need to use the from modulename import ... statement. This form of import provides us with the convenience of not having to specify the modulename, when accessing the imported attribute.

lename

ving to specify the modulename, when accessing the imported attribute. So no more `random.randint()` calls, only `randint()` . And make no mistake, here again the entire code of imported file is executed. Just the requested attribute name is copied into the scope of the importing program.

There are three ways, how we can import attributes in Python - import single, multiple, or all attributes.

Importing single attribute

```
>>> from test import var1
Print is being executed
>>> var1
52
>>> test.var2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'test' is not defined
```

Importing multiple attributes

We can import multiple attributes in one import statement. This is also in accordance with the official style guide. It saves space and typing.

```
>>> from test import var1,var2
Print is being executed
>>> var1
52
>>> var2
53
>>> func(var1,var2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

100% z Lekce 12

Importing all attributes

Importing all attributes

Osnova

In this case we use the asterisk instead of names (operator *):

```
>>> from test import *
Print is being executed
>>> var1
52
>>> var2
53
>>> func(var1,var2)
105
```

Here the asterisk means that we **copy all** the attribute names into the scope of importing program. This way of importing is often discouraged. The reason is that the person who will read the code will not know, which names have been imported and which not. And which names pertain to which module - imagine that multiple such statements would be listed in the script. It would be very difficult to understand the code.

We will discus importing with asterisk futher in the following unit - Data Hiding.

100% z Lekce 12

Data Hiding

By data hiding we refer to the fact, that the programmer can specify, what module attributes cannot be imported using `from module import *` syntax. This specification can have 2 forms:

1. Variable names beginning with a **single underscore**
2. Using `__all__` variable

Single Underscore Variable

We have added to our `test.py` script a new variable called `_secret` :

```
_secret = 'Password123'
print('Print is being executed')
var1 = 52
var2 = 53

def func(v1,v2):
    result = v1 + v2
    return result
```

Variable `_secret` contains a single underscore in front of it. This tells Python, that we do not want it to be imported, if the `from module import *` statement is used.

If we prform the import, we see, that we cannot access `_secret` variable. It has not been imported.

```
>>
   Osnova
2
>>> var2
53
>>> _secret
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_secret' is not defined
```

However, if whole module or the attribute `_secret` are imported, the hiding effect is gone:

```
>>> from test import _secret
Print is being executed
>>> _secret
'Password123'
```

```
>>> import test
Print is being executed
>>> test._secret
'Password123'
```

Variable all

If a module contains `__all__` variable, only those variable names will be imported, that are included in the list of strings referenced by `__all__` variable. That is, `__all__` **has the converse effect than the undescored names.**

```python
_secret = 'Password123'
print('Print is being executed')
var1 = 52
var2 = 53

def func(v1,v2):
    result = v1 + v2
    return result

```

100% z Lekce 12

```
>>> from test import *
Print is being executed
>>> var1
52
>>> var2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var2' is not defined
>>> _secret
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_secret' is not defined
```

```
>>> from test import *
Print is being executed
>>> var1
52
>>>
```

If `__all__` references an empty list, then nothing is imported:

```python
_secret = 'Password123'
print('Print is being executed')
var1 = 52
var2 = 53

def func(v1,v2):
    result = v1 + v2
    return result

__all__ = []
```

```
>>> from test import *
Print is being executed
>>> var1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var1' is not defined
```

And if only the underscored variable is listed in `__all__`, then only this one is imported:

```python
_secret = 'Password123'
print('Print is being executed')
```

100% z Lekce 12

```
Osnova

def func(v1,v2):
    result = v1 + v2
    return result

__all__ = ['_secret']
```

```
_secret = 'Password123'
print('Print is being executed')
var1 = 52
var2 = 53

def func(v1,v2):
result = v1 + v2
return result

__all__ = ['_secret']
```

```
>>> from test import *
Print is being executed
>>> _secret
'Password123'
```

If `__all__` is not specified, then `from module import *` imports all the variable names excluding those with leading single underscore.

Importing as

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES + / IMPORTING AS

Let's say, we would like to import a single attribute from a module, but that attribute has the same name as a variable, that is already present in our program. What can we do?

1. Change the name of the variable in our program
2. Import the whole module, containing the required variable and access it through `module.attribute` notation
3. Use aliasing - `as`

Aliasing will avoid potential name clashes thanks to aliasing the original attribute name. We can alias not only attribute but the module name as well. We often use module or attribute aliasing, when the module name and the attribute names are very long. Thus we save ourselves some typing.

100% z Lekce 12

Example of avoiding name clashes:

Osnova

```
>>> var1 = 1
>>> from test import var1 as v1
Print is being executed
>>> var1
1
>>> v1
52
```

To save typing, when invoking Python's `pprint()` (pretty print) function located inside `pprint` module, we write the following import statement:

```
>>> from pprint import pprint as pp
....
 '__spec__': ModuleSpec(name='test', loader=
<_frozen_importlib_external.SourceFileLoader object at 0x7f54999e4278>,
origin='/home/martin/PythonBeginner/Lesson8/test.py'),
 'func': <function func at 0x7f549b532bf8>,
 'var1': 52,
 'var2': 53}
```

Pretty print is used to neatly display complex dictionary structures.

Module aliasing

The `datetime` module has quite a long name as well as attributes it contains. Therefore we could import it as follows:

```
>>> import datetime as dt
```

And then we can create a datetime object with much less typing:

```
>>> dt.datetime(2020,1,1)
datetime.datetime(2020, 1, 1, 0, 0)
```

100% z Lekce 12

Osnova

Reload Module

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES + / RELOAD MODULE

Sometimes we run a script that imports a module that we have changed during the script runtime. We do not want to terminate the script, however we would like the changes to be reflected in it. Using the import statement will not help us, because Python will return already imported module object.

To solve this situation Python offers us `reload()` function that exists in the `importlib` standard library module.

We can import it in 2 ways:

```python
import importlib
importlib.reload(modulename)
```

```python
from importlib import reload
reload(modulename)
```

The modulename has to be a name of a module, that has already been imported. Otherwise we will get `NameError` .

We can get an overview of imported modules by checking the `sys.modules` variable.

The reload function re-runs the module source code and changes the imported module object in place. That way, the change affects all the references to the module object in your code. However, every variable that references to attributes imported using `from modulename import` statement will still point to attribute values before the reload was performed - reload does not affect those attribute values.

For example, we have the following `test.py` script:

```python
_secret = 'Password123'
print('Print is being executed')
var1 = 52
var2 = 53
```

```
_secret = 'Password123'
print('Print is being executed')
var1 = 52
var2 = 53

def func(v1,v2):
 result = v1 + v2
 return result

__all__ = ['_secret']
```

100% z Lekce 12

```
         lt    v1 + v2
 Osnova  rn   sult

__all__ = ['_secret']
```

We import it into interactive console to test the `func()` function:

```
>>> import test
Print is being executed
>>> test.func(1,2)
3
```

Now we change the addition operation to multiplication in the `func()` inside the `test.py` module:

```
...
def func(v1,v2):
    result = v1 * v2
    return result
...
```

And we reload the `test` module:

```
>>> from importlib import reload
>>> reload(test)
Print is being executed
<module 'test' from '/home/martin/PythonBeginner/Lesson8/test.py'>
>>> test.func(1,2)
2
```

We can see the change has been reflected in the `func` behaviour.

Compilation to byte code

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES + / COMPILATION TO BYTE CODE

100% z Lekce 12

When Python imports a module, that has never been imported before, it compiles it first into so called bytecode.

- **Compilation** is transformation of a source code into its binary representation - represenation in 0s and 1s that computers understand better.

- **Bytecode** is binary representation of Python script.

Compiled files are appended `.pyc` suffix and stored inside the `__pycache__` directory situated in the same folder as the original .py file.

If original file was called hack.py and was imported for example under Python 3.5., then the compiled file will be called hack.cpython-35.pyc. So the logic, how Python creates the .pyc file name is as follows:

```
original_filename.python_implementation-pythonversion.pyc
```

As the compilation is expensive operation, Python tries to do it as few times as possible.Therefore, when hit the import statement, Python always decides, whether compilation should take place.

Python compiles .py files into byte code .pyc files under two conditions:

1. the file is imported for the first time - there is no corresponding .pyc file compiled under corresponding Python version, yet.
2. always, when the original source .py file has been changed. Do determine, whether a file should be recompiled, Python compares timestamps of both - if the .py file has more recent timestamp (was changed recently), the .pyc file is recompiled.

When importing, python is actually loading the .pyc file.

Python Math Library

PYTHON ACADEMY / 10. IMPORTING / IMPORTING MODULES + / PYTHON MATH LIBRARY

100% z Lekce 12

Math library contains functions, that calculate things like, factorial, logarithms etc. In short, it provides us with functionality to perform mathematical calculations.

Functions you may be interested in are:

Function	Description
`math.sqrt(x)`	Return the square root of x.
`math.factorial(x)`	Return x factorial. Raises ValueError if x is not integral or is negative.
`math.ceil(x)`	Return the ceiling of x, the smallest integer greater than or equal to x.
`math.floor(x)`	Return the floor of x, the largest integer less than or equal to x.
`math.gcd(a, b)`	Return the greatest common divisor of the integers a and b. If either a or b is nonzero, then the value of gcd(a, b) is the largest positive integer that divides both a and b. gcd(0, 0) returns 0.
`math.isfinite(x)`	Return True if x is neither an infinity nor a NaN, and False otherwise. (Note that 0.0 is considered finite.)
`math.isinf(x)`	Return True if x is a positive or negative infinity, and False otherwise.
`math.isnan(x)`	Return True if x is a NaN (not a number), and False otherwise.

Besides functions, math offers also value representation of some important constants:

Constant	Description
`math.pi`	The mathematical constant π = 3.141592..., to available precision.

100% z Lekce 12

Osnova stant	Description
`math.inf`	A floating-point positive infinity - equivalent to the output of `float('inf')` . For negative infinity, use `-math.inf`

To find out, what other functions are contained within the math library see Python standard library documentation.

OS MODULE

What for os module?

100% z Lekce 12

This module is usually used for managing of file system content (files and directories) as well as creation and control of processes (programs).

The `os` module is very handy, when we want to:

- Work with files and directories - e.g create, remove, replace, rename them
- Examine the file system contents - list directories, walk the file system tree
- Get the information related to file status - when it has been last accessed, who accessed it etc.

Very important part of the os package is the `os.path` module, which provides functions to manipulate file paths.

In the next few sections, we will demonstrate the work with `os` on concrete examples.

List the Directory Content & CWD

PYTHON ACADEMY / 10. IMPORTING / OS MODULE / LIST THE DIRECTORY CONTENT & CWD

Before you write any of the commands below, import the module first: `import os`

In the following few units dedicated to os module, we will often need to list the content of a directory. Many times it will be the current working directory (CWD). To do that, we will use os module's `listdir()` function. By default, this function lists the content of CWD:

```
>>> os.listdir()
[TestDir', 'lesson8.md', 'test.py']
```

The result returned by the `listdir()` function depends on the content of your CWD. We should note, that **the directory content is listed in a list**.

To list the content of a directory different than CWD, we should pass into `listdir()` a string representing path to that directory:

```
>>> os.listdir('C:\\Windows\\System32')
```

```
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx',
...,
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW',
'zipfldr.dll']
```

100% z Lekce 12

Osnova

```
xw   ,   wtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

Note that we use **double backslah** inside the path string. We have to do such escaping on Windows, in order Python interprets the second backslash literal backslash and not as a part of a special character.

To determine what the CWD is, we use `os.getcwd()` , which returns a string representing path, to the CWD:

```
>>> os.getcwd()
'C:\\Users\\Program Files'
```

Creating Files and Directories

PYTHON ACADEMY / 10. IMPORTING / OS MODULE / CREATING FILES AND DIRECTORIES

Before we create any files, we should **create a new folder**. To do that, we need to import the os module and use its function `mkdir()` to make a new directory. The function expects one mandatory argument - **path** - place where the new directory should be created. In the example below, we provide relative path and new directory will be created in current working directory (`Lesson8`):

```
>>> os.mkdir('TestDir')
```

Now we should list the CWD to check, whether the folder `TestDir` has been created:

```
>>> os.listdir()
['TestDir', 'lesson8.md', 'test.py']
```

To **create a new file**, we can use the built-in `open()` function, opening the file in `'w'` mode. Let's create a new file inside the `TestDir` directory. We name the file `test.txt` :

```
>>> with op
        f.w
```

```
>>> with open('TestDir/test.txt','w') as f:
... f.write('This is a test file')
>>> os.listdir('TestDir')
['test.txt']
```

```
>>          ir('TestDir')
      Osnova
 t
```

Code Tasks

1. In directory `Lesson8` create two other directories: `TestDir2`, `TestDir3`
2. Inside the directory `TestDir`, create two new files: `tasks.txt`, `config.txt`

After completing this task, directory `Lesson8` should contain the following hierarchy of files and folders:

```
Lesson8
 |____TestDir
        |____config.txt
        |____tasks.txt
        |____test.txt
 |____TestDir2
 |____TestDir3
 ... and your original files and folders ...
```

Moving, Renaming and Replacing

PYTHON ACADEMY / 10. IMPORTING / OS MODULE / MOVING, RENAMING AND REPLACING

In the previous section, we have created new files and folders inside `Lesson8` as follows:

```
Lesson8
 |____TestDir
        |____config.txt
        |____tasks.txt
        |____test.txt
 |____TestDir2
 |____TestDir3
```

100% z Lekce 12

Wher to name, replace, or move files & folders we use `os.rename(source,`

Osnova

When we want to rename, replace, or move files & folders we use os.rename(source, destination) . The source is a path to the file we want to rename. The parameter destination specifies the new name of renamed file or folder.

So to **change name** of a file or a folder:

```
>>> os.rename('TestDir', 'TestDir1')
```

The `destination` parameter allows us not only to change the name of the file/folder, but also to move it to a new location. We can test this on file `tasks.txt` . We will rename it to `todo.txt` . We **move** it to the folder `TestDir2` :

```
>>> os.rename('TestDir1/tasks.txt','TestDir2/todo.txt')
>>> os.listdir('TestDir1')
['test.txt', 'config.txt']
>>> os.listdir('TestDir2')
['todo.txt']
```

We can do the same with directories:

```
>>> os.mkdir('TestDir1/InnerDir')
>>> os.rename('TestDir1/InnerDir', 'TestDir2/InnerDir2')
>>> os.listdir('TestDir2')
['InnerDir2', 'todo.txt']
```

Our `Lesson8` folder looks now like this:

```
Lesson8
 |____TestDir1
         |____config.txt
         |____test.txt
 |____TestDir2
         |____InnerDir2
         |____todo.txt
 |____TestDir3
 ... and your original files and folders ...
```

100% z Lekce 12

Osnova

l Module

PYTHON ACADEMY / 10. IMPORTING / OS MODULE / A BIT OF SHUTIL MODULE

Meanwhile os module allows us to move, rename and replace a file, we are not able to keep the original file in the original location and just copy its content into the new location. Therefore, for completeness, we include here demonstration of the `copy` functions that pertain to **shutil module**:

`shutil.copy(source,destination)`

```
>>> shutil.copy('TestDir1/config.txt','TestDir3/config3.txt')
'TestDir3/config3.txt'
>>> os.listdir('TestDir3')
['config3.txt']
>>> os.listdir('TestDir1')
['test.txt', 'config.txt']
```

Our `Lesson8` folder after the above change:

```
Lesson8
 |____TestDir1
         |____config.txt
         |____test.txt
 |____TestDir2
         |____InnerDir2
         |____todo.txt
 |____TestDir3
         |____config3.txt
 ... and your original files and folders ...
```

Removing Files and Directories

PYTHON ACADEMY / 10. IMPORTING / OS MODULE / REMOVING FILES AND DIRECTORIES

100% z Lekce 12

Osnova

```
as
|____        1
        |____config.txt
        |____test.txt
|____TestDir2
        |____InnerDir2
        |____todo.txt
|____TestDir3
        |____config3.txt
... and your original files and folders ...
```

To remove a file, use `os.remove()` . When trying to remove a nonexistent file, we get the `FileNotFoundError` :

```
>>> os.remove('TestDir3/config3.txt')
>>> os.remove('TestDir3/config3.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'TestDir3/config3.txt'
```

We cannot remove a directory with `os.remove()` . To remove a directory, use `os.rmdir()` :

```
>>> os.remove('TestDir3')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IsADirectoryError: [Errno 21] Is a directory: 'TestDir3'
>>> os.rmdir('TestDir3')
```

In case we would like to remove non-empty directory, we need to use the module called `shutil` and its function `rmtree()` :

```
>>> import shutil
>>> shutil.rmtree('TestDir1')
```

But let's not remove the folder `TestDir1` .

100% z Lekce 12

```
|   es
|____   Osnova  1
        |____config.txt
        |____test.txt
|____TestDir2
        |____InnerDir2
        |____todo.txt
... and your original files and folders ...
```

Manipulating Paths

PYTHON ACADEMY / 10. IMPORTING / OS MODULE / MANIPULATING PATHS

Why is good to know, how paths can be manipulated? Many times, when trying to locate files or folders in Python programs, we find ourselves in the following situations:

- We have a list of folders and we would like to join them into a path or vice versa - we have a path, and would like to get a list of folders.
- We would like to extract only folder or file names from a path

What does path manipulation mean?

Path is a string representing location of a file or a directory in a filesystem. Paths on different operating systems may look differently. For example paths in Windows separate folder using backslash `'\'` and in Linux and MacOS, using forward slash `'/'` . Also the root directory looks differently on Windows vs. Linux/Mac. Linux and Mac use `/` for root directory and Windows have for example C root `C:` .

Therefore, performing the path extraction or path creation has to take into consideration OS specifics and module `os.path` makes this easier for Python programmers.

Path Creation

100% z Lekce 12

ıld ''' te ath from them:
Osnova

```
>>> items = ['/', 'usr', 'PythonBeginner', 'Lesson8', 'test.txt']
>>> os.path.join(*items)
'/usr/PythonBeginner/Lesson8/test.txt'
```

On Windows, the above result would look like:

```
>>> items = ['C:', os.sep, 'User', 'PythonBeginner','Lesson8','test.txt']
>>> os.path.join(*items)
'C:\\User\\PythonBeginner\\Lesson8\\test.txt'
```

Expression `os.sep` evaluates to path separator used in the given operating system. On Windows it is a backslash `\`. Unfortunatelly, we have to put it there in order to separate correctly the disc name from the rest of the names in the path.

Path Decomposition

Let's say we have the following path:

```
>>> path = '/usr/PythonBeginner/Lesson8/test.txt'
           |___________________________|________|
                         |                  |
                      dirname        basename/filename
```

If we would like to **extract the** `dirname` (path consisting of directories only) part, we perform the following call:

```
>>> os.path.dirname(path)
'/usr/PythonBeginner/Lesson8'
```

To **extract only the** `filename`, we use:

```
>>> os.path.basename(path)
'test.txt
```

To get **both** at the same time, we use `os.path.split`. In other words `os.path.split(path)`

100% z Lekce 12

```
>>>      sp  (path)
   Osnova
         nner/Lesson8', 'test.txt')
```

And finally to **extract only the file extension** we use `os.path.splitext()` :

```
>>> os.path.splitext(path)
('/usr/PythonBeginner/Lesson8/test', '.txt')
```

Checking Path Validity

PYTHON ACADEMY / 10. IMPORTING / OS MODULE / CHECKING PATH VALIDITY

To check, whether we have been provided a valid path, we can use the following three `os.path` functions:

- `os.path.exists(path)` - returns `True` , if a given path exists in the file system:
- `os.path.isfile(path)` - returns `True` , if a given path references a file
- `os.path.isdir(path)` - - returns `True` , if a given path references a directory

Let's demonstrate the working principles using our `Lesson8` folder:

```
Lesson8
 |____TestDir1
          |____config.txt
          |____test.txt
 |____TestDir2
          |____InnerDir2
          |____todo.txt
 ... and your original files and folders ...
```

We should set `Lesson8` to be our current working directory (CWD) - we should move into it. Once we are there, we will extract the CWD:

```
>>> CWD = os.getcwd()
```

100% z Lekce 12

[illegible] the below checks using path stored inside `CWD`. Check, whether
[illegible] Osnova [illegible] `xt'` **exists**:

```
>>> path = os.path.join(CWD,'TestDir1/config.txt')
>>> path
'path_to_lesson8/Lesson8/TestDir1/config.txt'
>>> os.path.exists(path)
True
```

Check, whether `'TestDir1/config.txt'` **is a file**:

```
>>> path = os.path.join(CWD,'TestDir1/config.txt')
>>> os.path.isfile(path)
True
```

Check, whether `'TestDir2/InnerDir2'` **is a directory**:

```
>>> path = os.path.join(CWD,'TestDir2/InnerDir2')
>>> os.path.isdir(path)
True
```

Getting File Size

PYTHON ACADEMY / 10. IMPORTING / OS MODULE / GETTING FILE SIZE

Let's say we have created a file `TestDir1/test.txt` in the past.

```
>>> with open('TestDir1/test.txt','w') as f:
...     f.write('This is a test file')
```

To find out size of a file at a given path, we use `os.path.getsize()` :

```
>>> os.path.getsize('TestDir1/test.txt')
19
```

100% z Lekce 12

OS MODULE +

Walking the File System

We can use function `os.walk` to iterate over all the directories and filenames under a given path. The `os.walk` function returns on each iteration a tuple `(dirpath, dirnames, filenames)` where:

- `dirpath` is a directory, that is currently being visited
- `dirnames` - list of directory names inside the currently visited directory
- `filenames` - list of file names inside the currently visited directory

An example of walking our `Lesson8` directory

```
|        '      c  fig.txt
       Osnova
         ____t   .txt
 |____TestDir2
        |____InnerDir2
        |____todo.txt
 ... and your original files and folders ...
```

would be:

```
>>> CWD = os.getcwd()
>>> for dirpath, dirnames, filenames in os.walk(CWD):
...     print(dirpath.upper())
...     print('\nDIRNAMES')
...     print(dirnames)
...     print('\nFILENAMES:')
...     print(filenames)
...     print('=' * 20)

...
/HOME/USER/PYTHONBEGINNER/LESSON8
DIRNAMES
[TestDir1', 'TestDir2', ..and your original folders]
FILENAMES:
[... your original files...]
====================
/HOME/USER/PYTHONBEGINNER/LESSON8/TESTDIR1
DIRNAMES
[]
FILENAMES:
['test.txt', 'config.txt']
====================
/HOME/USER/PYTHONBEGINNER/LESSON8/TESTDIR2
DIRNAMES
['InnerDir2']
FILENAMES:
['todo.txt']
```

100% z Lekce 12

Osnova

```
[]
====================
... and your original files and folders ...
```

Running Shell Commands

PYTHON ACADEMY / 10. IMPORTING / OS MODULE + / RUNNING SHELL COMMANDS

The os module provides us with very interesting possibility to issue shell commands inside running Python script - `os.system()` . By shell commands we mean those commands, that we write to terminal to change (`cd`) list a directory (`dir` on Windows resp. `ls` on Linux). A handy command is the one, that clears the terminal screen - `clear` on Linux/Mac and `cls` on Windows:

Clearing the screen on Linux/Mac:

```
>>> os.system('clear')
```

or on Windows:

```
>>> os.system('cls')
```

The `os.system()` function expects commands to be passed as strings.

Getting the Terminal Size

PYTHON ACADEMY / 10. IMPORTING / OS MODULE + / GETTING THE TERMINAL SIZE

In case you will create an application that communicates with the user via terminal, you may

100% z Lekce 12

```
 >>          r    al_size()
      Osnova
 s.          _si.  columns=80, lines=24)
>>> os.get_terminal_size().columns
80
>>> os.get_terminal_size().lines
24
```

Number of columns tells us, how many characters can be written on one line and number of lines tells, us, how many rows are visible on a terminal at once. So to print a line that will take exactly one row, we do as follows:

```
>>> cols = os.get_terminal_size().columns
>>> cols
80
>>> print('-'*cols)
-------------------------------------------------------------------------
-------
>>>
```

Moving with shutil

PYTHON ACADEMY / 10. IMPORTING / OS MODULE + / MOVING WITH SHUTIL

`shutil.move(source,destination)`

The advantage of this function compared to `os.rename()` is that it copies all the directories that the moved directory contains. To demonstrate this, we will create multiple directories nested within each other using `os.makedirs()` function:

```
>>> os.makedirs('TestDir1/Inner1/InInner1')
```

Let's say that our `Lesson8` folder looks as follows:

```
Lesson8
```

100% z Lekce 12

```
              _InInner1
      Osnova
          _c    ig.txt
        |____test.txt
  |____TestDir2
        |____InnerDir2
  |____TestDir3
```

Now we will move the whole `Inner1` directory and its subdirectories into `TestDir3` :

```
>>> shutil.move('TestDir1/Inner1','TestDir3/Inner3')
'TestDir3/Inner3'
>>> os.listdir('TestDir3/Inner3')
['InInner1']
>>> os.listdir('TestDir1')
['test.txt', 'config.txt']
```

We can see that `TestDir1` does not contain the folder `Inner1` anymore. It was moved to `TestDir3` under new name `Inner3` .

100% z Lekce 12

Osnova

What do we already know?

PYTHON ACADEMY / 10. IMPORTING / SYS MODULE BASICS / WHAT DO WE ALREADY KNOW?

In this lesson, we have already learned about two variables that are used when importing modules - `sys.modules` and `sys.path` .

In the previous lesson, we use `sys.exc_info()` function to access information related to exception.

We have also seen, that `sys.stdout` represents terminal window, to which the information is printed.

Determine Operating System

PYTHON ACADEMY / 10. IMPORTING / SYS MODULE BASICS / DETERMINE OPERATING SYSTEM

We can determine the operating system, that our script is currently running on, by using `sys.platform` . Based on the system the values returned can be as follows:

System	Value
Linux	'linux'
Windows	'win32'
Mac OS X	'darwin'

Why is this useful? In the previous section on os module, we have seen the command `os.system()` that enables us to run shell commands inside Python script. Windows and

100% z Lekce 12

' 'he ' ' ' 'lo' ve want to clear the terminal window, but as the command is different for
Osnova
ia ux, c, we have to check the `sys.platform` , before issuing it:

```
>>> command = 'cls' if sys.platform.startswith('win') else 'clear'
>>> os.system(command)
```

Collect Command Line Arguments

PYTHON ACADEMY / 10. IMPORTING / SYS MODULE BASICS / COLLECT COMMAND LINE ARGUMENTS

This feature makes our programs much more user friendly. It allows us to send data into Python script from the command line, where we launch the script.

Command line arguments are additional options we pass following the name of the launched script. For example the script below is given two arguments - `Lesson6` and `Lesson7` . As the script name is `move_files.py` we can guess, that use wants to run a script, that will move files from the directory Lesson6, to directory Lesson7.

```
~/PythonBeginner/Lesson8 $ python move_files.py Lesson6 Lesson7
```

We do not have to implement the above described functionality, but we can however check, what the variable `sys.argv` contains by writing the following line into the empty `move_files.py` file:

```python
import sys

print(sys.argv)
```

After running the above code we get the following output:

```
Le~/PythonBeginner/Lesson8 $ python move_files.py Lesson6 Lesson7
['move_files.py', 'Lesson6', 'Lesson7']
```

The first item in the sys.argv list will always be a string containing the program's filename (`'move_files.py'`) and the rest are command line arguments. We can thus send our script

100% z Lekce 12

Osnova ram

To exit from Python program without having to wait until the execution reaches the last line, we can call `sys.exit()` . Call to this function raises the `SystemExit` exception, so cleanup actions specified by finally clauses of try statements are honored, and it is possible to intercept the exit attempt as well:

```
try:
    sys.exit()
except SystemExit:
    inpt = input('Do you really want to exit the program? y/n ')
    if inpt == 'y': raise
```

QUIZ

100% z Lekce 12

) Osnova

PYTHON ACADEMY / 10. IMPORTING / QUIZ / IMPORTING

1/15

What is a **module**?

A. Arithmetic operator

B. Any Python file

C. Describes, whether a file is writable or readable

D. Special Python version

os & sys Modules

PYTHON ACADEMY / 10. IMPORTING / QUIZ / OS & SYS MODULES

1/18

Which of the below commands creates a directory `'Home'` ?

A. os.makedir('Home')

100% z Lekce 12

B. os.mkdir('Home')

Osnova

C. os.dir('Home')

D. makedir('Home')

HOME EXERCISES

Checking Already Imported Modules

PYTHON ACADEMY / 10. IMPORTING / HOME EXERCISES / CHECKING ALREADY IMPORTED MODULES

100% z Lekce 12

1. I tic pprint from module pprint

Osnova

2. Pass to the function pprint the dictionary of currently imported modules into the Python interpreter.

How you will do that?

Example of how the result could look like:

```
{'__main__': <module '__main__' (built-in)>,
 '_bootlocale': <module '_bootlocale' from
'/home/martin/anaconda3/lib/python3.5/_bootlocale.py'>,
 '_codecs': <module '_codecs' (built-in)>,
...
 'types': <module 'types' from
'/home/martin/anaconda3/lib/python3.5/types.py'>,
 'weakref': <module 'weakref' from
'/home/martin/anaconda3/lib/python3.5/weakref.py'>,
 'zipimport': <module 'zipimport' (built-in)>}
```

3. Print only the module names, each on a new line (you do not have to use pprint):

Example of how the result could look like:

```
encodings.latin_1
itertools
encodings.aliases
encodings.cp437
_signal
...
encodings.utf_8
io
os.path
_sysconfigdata
_sre
```

Code Solution

100% z Lekce 12

Osnova ... ution

Using pprint we can print each dictionary record on new line (any nested dictionary is indented) as follows:

```
from pprint import pprint
import sys
pprint(sys.modules)
```

To print only key names, each on new line we can use the following trick:

```
print('\n'.join(sys.modules.keys()))
```

We join all the keys on `\n` newline character, that will print each key on a new line. If we did not pass the joined string into the `print` function, the result would look like this:

```
'encodings.latin_1\nitertools\nencodings.aliases\nencodings.cp437\n_signal
```

Find Standard Library Modules

PYTHON ACADEMY / 10. IMPORTING / HOME EXERCISES / FIND STANDARD LIBRARY MODULES

Now, you task is pretty simple - `find`, where the folder containing `standard library` is located on your computer. Try it on your own or google it. Please, do not check the solution before trying on your own. :)

Code Solution

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

100% z Lekce 12

Click to see our solution

Osnova

Standard library modules are stored inside the directory called `Lib` or `lib` in your Python installation folder. Maybe the easiest way how to find this folder is to list all the paths that are searched during the module import. This list is stored inside the variable `sys.path` . If I pretty print the content of the `sys.path` variable, I realize that my `lib` folder is located at `'/home/martin/anaconda3/lib'`

```python
from pprint import pprint
pprint(sys.path)
```

This is what I see on my terminal.

```
['',
 '/home/martin/anaconda3/lib/python35.zip',
 '/home/martin/anaconda3/lib/python3.5',
 '/home/martin/anaconda3/lib/python3.5/plat-linux',
 '/home/martin/anaconda3/lib/python3.5/lib-dynload',
 '/home/martin/anaconda3/lib/python3.5/site-packages',
 '/home/martin/anaconda3/lib/python3.5/site-packages/PyDispatcher-2.0.5-
py3.5.egg',
 '/home/martin/anaconda3/lib/python3.5/site-packages/Sphinx-1.3.5-
py3.5.egg',
 '/home/martin/anaconda3/lib/python3.5/site-packages/setuptools-20.3-
py3.5.egg']
```

Find 3rd Party Modules

PYTHON ACADEMY / 10. IMPORTING / HOME EXERCISES / FIND 3RD PARTY MODULES

Find, where the folder containing `3rd party modules` is located on your computer. Try it on your own or google it. Please, do not check the solution before trying on your own. :)

100% z Lekce 12

below if you want to see, how we wrote the code.
Osnova

Click to see our solution

Third party modules are stored inside the directory called `site-packages` . Maybe the easiest way how to find this folder is to list all the paths that are searched during the module import. This list is stored inside the variable `sys.path` . If I pretty print the content of the `sys.path` variable, I realize that my `site-packages` are located at `'/home/martin/anaconda3/lib/python3.5/site-packages'`

```python
from pprint import pprint
pprint(sys.path)
```

This is what I see on my terminal.

```
['',
 '/home/martin/anaconda3/lib/python35.zip',
 '/home/martin/anaconda3/lib/python3.5',
 '/home/martin/anaconda3/lib/python3.5/plat-linux',
 '/home/martin/anaconda3/lib/python3.5/lib-dynload',
 '/home/martin/anaconda3/lib/python3.5/site-packages',
 '/home/martin/anaconda3/lib/python3.5/site-packages/PyDispatcher-2.0.5-
py3.5.egg',
 '/home/martin/anaconda3/lib/python3.5/site-packages/Sphinx-1.3.5-
py3.5.egg',
 '/home/martin/anaconda3/lib/python3.5/site-packages/setuptools-20.3-
py3.5.egg']
```

Import Hacks

PYTHON ACADEMY / 10. IMPORTING / HOME EXERCISES / IMPORT HACKS

100% z Lekce 12

working directory, we create a module of the same name as standard

Osnova

Inside the current working directory, we create a module of the same name as standard library module

1. Create a new file called `sys.py` inside the directory `Lesson 10`, and populate it with the following code:

```python
var1 = 1
var2 = 2

def func(a,b):
    result = a + b
    return result
```

2. Now open terminal, and set your current working directory to `Lesson 10` (move to that folder using `cd` command)
3. Once inside `Lesson 10` launch Python console:

```
Lesson10 $ python
Python 3.6.1 (default, Apr 20 2018, 16:51:17)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

4. Import module called `sys` :

```python
import sys
```

5. Now just write the name `sys` into the console and wait for the result:

```
>>> import sys
>>> sys
<module 'sys' (built-in)>
```

We can see that even though we have created module called `sys.py` inside the current working

100% z Lekce 12

6. N… …na… our `sys.py` to `random.py` and try to `import random`

Osnova

```
import random
```

7. Write the name `random` into the console. What will be the result?

```
>>> import random
>>> random
<module 'random' from '/home/martin/PythonAcademy/Lesson10/random.py'>
```

We can see, that this time, our own file has been imported. We can prove it by invoking `random.var1` :

```
>>> random.var1
1
```

Python has not found the `random` module in `sys.modules` dictionary and therefore had to search for module of that name in directories listed in `sys.path` . As the first directory searched was the current working directory, our own `random.py` has been imported.

Is a directory?

PYTHON ACADEMY / 10. IMPORTING / HOME EXERCISES / IS A DIRECTORY?

Write a function called `which` that will `check if a file path leads to a file or a directory` . The function should take one input - examinated path string.

Example use on Linux (or MacOS):

```
>>> which('/home/user/Downloads/python.pdf')
file
```

Example of use on Windows (different path separators and root folder):

```
>>> which(r'C:\Program Files')
```

100% z Lekce 12

Osnova

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

Click to see our solution

To solve this problem, we have to have knowledge of `os` module. Therefore, what our function will need to have available this module in the namespace. We therefore need to first import `os`.

```python
import os

def which(path):

    #check if path leads to a directory
    if os.path.isdir(path):
        return 'dir'

    # check if path leads to a file
    elif os.path.isfile(path):
        return 'file'
```

Our function makes use of two `os.path` module functions:

- `os.path.isdir(path)`
- `os.path.isfile(path)`

The above function behaves as follows:

```
>>> import os
>>> which('/usr/bin/env')
'file'
>>> which('/usr/bin')
'dir'
>>> which('/user/bin')
>>>
```

100% z Lekce 12

Let's say we have a directory called **DOCUMENTS** and inside this directory, we have files and other directories (those directories may have other files and directories inside themselves:

```
DOCUMENTS
    |____PICTURES
    |____MUSIC
    |____REPORTS
        |____solutions.doc
        |____results.pdf
        |____salaries.pdf
    |____PRESENTATIONS
        |____results.pdf
    |____solutions
        |___good.txt
        |___bad.txt
    |____info.txt
    |____solutions.doc
    |____passwords.txt
```

We would like to create a program, which we will give a name of a file or a directory and the name of the directory where to begin its search. The program will print and then return the list of all matches found inside a given directory to be searched.

The inputs should be passed into the program via command line arguments. That means, when launching the script from the command line.

So for example, if we gave our program name `solutions` and we tell it to start looking for such file or folder inside the directory called `DOCUMENTS` , we should get the following report:

```
Lesson10 $ python search.py /home/martin/PythonAcademy/Lesson10/DOCUMENTS
solutions
FILE : /home/martin/PythonAcademy/Lesson10/DOCUMENTS/solutions.doc
```

Osnova ǝ ɔlder name is passed to the script as absolute path (/home/martin/PythonAcademy/Lesson10/DOCUMENTS), in order Python knows exactly, where to begin its search.

Report printed by the program tells us, where the file or folders of a given name were found. It also gives us information, whether a given path represents a file or a directory.

And now it is your turn to implement the above program.

Code Solution

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

Click to see our solution

```
import sys,os


def search(start_dir, searched_name):
    paths = []
    dirs_to_search = [start_dir]
    item_type = ('dir','file')

    while dirs_to_search:
        current_dir = dirs_to_search.pop(0)

        for item in os.listdir(current_dir):
            item_path = os.path.join(current_dir,item)

            if item.split('.')[0] == searched_name:

                _type = item_type[os.path.isfile(item_path)]
                paths.append('{} : 
{}'.format(_type.upper(),item_path))
```

100% z Lekce 12

```
                    dirs_to_search.append(item_path)
```

Osnova

```
    print(*paths, sep='\n')
    return paths



if __name__=='__main__':

    try:
        start_dir, searched_name = sys.argv[1:]
    except ValueError:
        num_arguments = 'Too few' if len(sys.argv) < 3 else 'Too 
many'
        print(num_arguments+ ' arguments')
        print('Arguments exected in format: search.py START_FOLDER 
SEARCHED_NAME')
    else:
        search(start_dir, searched_name)
```

The first part of our script defines a function `search()` and he second part enclosed inside `if __name__=='__main__':` contains code, that is executed, only if the file is executed as Python script (not imported). We can thus import function search into another python program, without invoking the second part of the script.

Imports

We will use some functionality from modules `sys` and `os.path` , therefore the first line of the script contains their import statement:

```
import sys, os
```

Runnint the script

This part of the script serves to parse the inputs from the command line. It is only logical to include this kind of code unde this condition as the information is provided from command

100% z Lekce 12

W[illegible] arguments from the variable `sys.argv` . Argument at index 0 refers to the name [illegible] and therefore we are intereseted only in inputs at index 1 and 2.

Osnova

```python
if __name__=='__main__':

    try:
        start_dir, searched_name = sys.argv[1:]
    except ValueError:
        num_arguments = 'Too few' if len(sys.argv) < 3 else 'Too many'
        print(num_arguments+ ' arguments')
        print('Arguments exected in format: search.py START_FOLDER SEARCHED_NAME')
    else:
        search(start_dir, searched_name)
```

We also take care of incorrect user inputs - if too few inputs are provided, the user is informed and the program is terminated:

```
$ python search.py /home/martin/PythonAcademy/Lesson10/Solved/DOCUMENTS
Too few arguments
Arguments exected in format: search.py START_FOLDER SEARCHED_NAME
```

The same is valid for the situation, when too many arguments are entered:

```
$ python search.py /home/martin/PythonAcademy/Lesson10/Solved/DOCUMENTS solutions 2
Too many arguments
Arguments exected in format: search.py START_FOLDER SEARCHED_NAME
```

If correct number of inputs is provided, we call the `search()` function.

Function search()

```python
def search(start_dir, searched_name):
    paths = []
    dirs_to_search = [start_dir]
    item_type = ('dir','file')

```

100% z Lekce 12

Osnova

```
        item in os.listdir(current_dir):
        item_path = os.path.join(current_dir,item)

        if item.split('.')[0] == searched_name:

          _type = item_type[os.path.isfile(item_path)]
          paths.append('{} : {}'.format(_type.upper(),item_path))

        if os.path.isdir(item_path):
          dirs_to_search.append(item_path)

    print(*paths, sep='\n')
    return paths
```

The first thing we do is to

1. create the variable `paths` where we will store the resulting list of paths, that match the `searched_name
2. create the variable `dirs_to_search` that will serve as the itinerary - which directories are to be visited still. The first directory to be visited is the one, we have entered on the command line. All those, that have been visited will be removed from the list, in order we do not return there anymore.
3. from the `item_type` tuple will extract the value `file` if the checked item in a directory is a file and `dir` if it is a directory

Then we enter the `while` loop, that will run while the list `dirs_to_search` is not empty.

Inside the loop, from the variable `dirs_to_search` , we will pop the next directory to be visited and store it inside the current directory variable.

Note that `dirs_to_search` contains a list of absolute paths to directories and so we can pass each path, we pop out from it as a input into `os.listdir()` function to iterate over the `current_dir` 's content. Iterating means, examining each item in the directory and checking, whether its name matches the `searched_name` and if so, then collect its path.

If the examined item in the current directory is another directory, we should store it in our `dirs_to_search` list, in order we can visit it in the future.

100% z Lekce 12

```
    item_type = ('dir','file')
    Osnova
    ...
            ...

            _type = item_type[os.path.isfile(item_path)]
```

In order we don't have to use condition to tell, whether an item is a file or directory, we have made use of the fact, that the function `os.path.isfile()` returns True (1), if the given path refers to a file and False (0) if not, Therefore we can use its return value as index to retrieve the corresponding flag - type - from the variable `item_type` .

If you are asking, why have we used underscore at the beginning of the variable name `_type` , it is because, there exists a built-in function `type` and we do not want to mix those two.

Mortgage Into Modules [H]

PYTHON ACADEMY / 10. IMPORTING / HOME EXERCISES / MORTGAGE INTO MODULES [H]

In Home Exercises in `Lesson 8` , we have created a script to calculate and list `mortgage payments` . Now, our task is to distribute some of its functions into separate modules in order we keep related functionality in one module.

Let's create three files, where will we store the functions as follows:

1. `calc.py` - here we will store functions, whose job is to calculate the payments
2. `display.py` - here we will store the functions that take care of nice formatting and printing the result to the terminal
3. `main.py` - here we will store functions that make use of functions stored in `calc.py` and `display.py` . Basically here should be stored only functions, that actually manage / orchestrate the whole program.

100% z Lekce 12

Osnova

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

Click to see our solution

First, we will create the three files in the same folder. Now, based on the solution from lesson 8, we have distributed the functions into individual files as follows:

File	Functions
`display.py`	`make_table` , `make_header` , `column_widths` , `extract_column` , `format_rows` , `insert_lines`
`calc.py`	`monthly_payment` , `total_mortgage` , `total_interest` , `payments`
`main.py`	`main` , `collect_inputs`

If we now ran the `main.py` , we end up with `NameError` , because the function `main` tries to call the function `make_table`

```
$ python main.py
How much do you want to borrow?: 2000000
At what rate?: 2.09
How many years?: 30
Traceback (most recent call last):
  File "main.py", line 70, in <module>
    main()
  File "main.py", line 14, in main
    table = make_table(pmts, header_inputs)
NameError: name 'make_table' is not defined
```

This means, we have to import the function `make_table` from `display.py` . We will add the import statement at the top of the file `main.py` :

```
from display import make_table
```

100% z Lekce 12

Nc ur s ipt `main.py` again and we get another `NameError` :

Osnova

```
Traceback (most recent call last):
  File "main.py", line 35, in <module>
    main()
  File "main.py", line 8, in main
    pmt = monthly_payment(loan,monthly_rate,num_periods)
NameError: name 'monthly_payment' is not defined
```

We still need to import `montly_payment` from `calc.py` into `main.py` :

```python
from display import make_table
from calc import monthly_payment

# the rest of main.py code
```

After running `main.py` we get yet another error:

```
Traceback (most recent call last):
  File "main.py", line 36, in <module>
    main()
  File "main.py", line 10, in main
    total_to_pay = total_mortgage(pmt,num_periods)
NameError: name 'total_mortgage' is not defined
```

It is time to look well at the `main` function. What else will it need to work properly besides the function `total_mortgage` ?

```python
...

    total_to_pay = total_mortgage(pmt,num_periods)
    interest = total_interest(total_to_pay, loan)
    pmts = payments(pmt,total_to_pay,monthly_rate,num_periods)


    header_inputs=(loan,num_periods,interest,pmt)

    table = make_table(pmts, header_inputs)
    print(table)
    with open('mortgage.txt', 'w') as file:
```

100% z Lekce 12

W[illegible] [illegible]ed [illegible]tal_interest , payments what actually covers all the functions we I[illegible] Osnova [illegible] [illegible]e calc.py . We should better change the original import from calc.py containing only monthly_payment into the whole module:

```python
import calc
from display import make_table

# the rest of main.py code
```

As we have imported the whole module and not individual functions, we will have to prepend all the function names, that belong to calc.py with calc. in the main.py script:

```python
def main():
    ....

    pmt = calc.monthly_payment(loan,monthly_rate,num_periods)
    total_to_pay = calc.total_mortgage(pmt,num_periods)
    interest = calc.total_interest(total_to_pay, loan)
    pmts = calc.payments(pmt,total_to_pay,monthly_rate,num_periods)
```

Now our script main.py works properly and we are done:

```
$ python main.py
How much do you want to borrow?: 2000000
At what rate?: 2.09
How many years?: 30
|  Loan: 2000000  |  Years: 30  |  Interest: 693880  |  Monthly Payment:
7483  |
==========================================================================
|     Payment     |  Interest  |     Principal     |        Left to Pay
|
==========================================================================
|        1        |   4,692    |       2,791       |         2,693,880
|
|        2        |   4,679    |       2,804       |         2,686,397
|
|        3        |   4,666    |       2,817       |         2,678,914
|
|        4        |   4,653    |       2,830       |         2,671,431
```

100% z Lekce 12

Osnova

script

To make our `main.py` script even better, we can enclose the call to `main()` function at the bottom of the file into the condition `if __name__ == '__main__'`:

```
if __name__ == '__main__':
    main()
```

DALŠÍ LEKCE