Theory: os module

28 minutes 0 / 5 problems solved

Skip this topic Start practicing

381 users solved this topic. Latest completion was about 5 hours ago.

Current topic:

os module

Topic depends on:

✓ Load module Stage 1 5★

✓ Writing files Stage 2

✗ Parameters and options Stage 1

Sometimes, you may need to interact with the current operating system and access its features when working on your programs. You may need to know whether it would be easy to run it on other systems. If you need to get the list of files and folders in the current working directory, you may know that there are different commands for this on Linux/macOS and Windows. Pathnames is another issue — there are different conventions on different platforms; the absolute path on Linux and macOS systems starts with a forward slash `/`, on Windows, it should start with a drive letter and a backslash `\`. In this case, your program should be prepared for that.

It is useful to have the tools that would work on any OS so that you wouldn't need to handle all possible outcomes manually. Python provides two related built-in modules: `os` for working with files and directories and `os.path` for handling paths from different OS systems. You can access both `os` and `os.path` by loading the module as:

```
import os
```

In the topic, we will discuss the basics of both modules.

Table of contents:

§1. Current working directory

Suppose we have a very simple program *divide.py* that writes the result of the division of two input numbers to a separate text file:

```
# divide.py
div_result = int(input()) / int(input())

file = open('division_result.txt', 'w', encoding='utf-8')
file.write(str(div_result))
file.close()
```

If we store *divide.py* in a separate PyCharm project, it may be located as `/home/user/PycharmProjects/project/divide.py`. However, depending on your operating system (especially on Windows), the path can look different.

In the example above, we did not specify the directory where we want to create the file, so when we open the `project` folder from the PyCharm IDE and launch *divide.py*, the program will write the text file with the result to the same `project` directory. It happens because we execute *divide.py* from the `project` folder, so it becomes the ***current working directory***. We can check the current working directory using the `os.getcwd()` function. The result is returned as a string:

```
print('The current working directory is', os.getcwd())
# The current working directory is /home/user/PycharmProjects/project
```

Now, assume that we want to launch the program from the command-line interpreter by typing the following statement:

```
python3 /home/user/PycharmProjects/project/divide.py
```

The text file will be created in the directory of the Python command-line interpreter. So, it is the home directory. We can similarly check the current working directory from CLI:

```
python3
# ...
# Type "help", "copyright", "credits" or "license" for more information.
>>>import os
>>>os.getcwd()
# /home/user
```

The resulting file will be created in different places, depending on the current working directory, the directory where the program is executed. If you don't know it, it can cause problems, so, please, bear it in mind.

§2. Changing the working directory

We can change the working directory manually to avoid confusion. The `os.chdir()` function can be used for this. It takes the absolute or a relative path as an argument that is basically our desired directory. In the following example, we pass the (absolute) home directory path as a string:

```
os.chdir('/home/user')
```

Once we passed the path to `chdir()`, we can call `getcwd()` again to make sure that the working directory was changed correctly:

```
print('The current working directory is', os.getcwd())
# The current working directory is /home/user
```

The OSError exception (or its subclass) will tell us that the path or filename is not correct.

§3. Creating directories

We may want to create new directory when working on a piece of code. There are two functions to create new directories — `os.mkdir()` and `os.makedirs()`.

- `os.mkdir()` is used to create a single directory. To do so, we should pass the name of the new folder or the full path to it — so that it will be created in the working directory or in another specified directory, respectively. An example below illustrates the latter, we just pass the string `some_new_project` to `mkdir()`:

  ```
  os.mkdir('some_new_project')
  ```

- `os.makedirs()` allows us to create nested directories in the specified path. Similarly, we can indicate the full path or the names of directories. This function is applied in the following example when creating `course`, `students`, and `year`; `year` is created within the `students` directory and `students` in its turn is created within `course`.

  ```
  os.makedirs('course/students/year')
  ```

If a defined directory already exists, FileExistsError will be raised.

§4. Folder content

When working with directories in `os`, it is very easy to learn about their contents or even change it. We have two functions for that, `os.listdir()` and `os.rename()`.

- `os.listdir()` returns a list of names of all files and folders in the given directory. If not specified, the function will return the list of names for the current working directory.

  ```
  print(os.listdir('course'))
  # ['student_list.txt', 'students', 'course_plan.txt']
  ```

  It is a very important function. It can be used when you need to process all files in a folder. Note, however, that it returns both file names and folder names, so to get a proper list of files, you'll need to choose only those that end with "*.txt*" in our example.

- `os.rename()` renames the file or the directory to the given name. Its first argument is the path to the text file, the name of which we want to change, and the second argument is the very same path, but with the new name, *list_of_students.txt* in our example.

```
os.rename('course/student_list.txt', 'course/list_of_students.txt')
```

§5. Access

With `os` you can also test whether the given path exists and learn whether it has special access rights. This functionality is implemented in the `os.access()`. Apart from the path, the function takes another argument, the ***mode*** value, which denotes an available way of operating or using a particular object on the system, e.g. reading or executing a file. We will cover all four possible mode values.

The mode `os.F_OK` is used to check the specified path for existence. If we create a new directory and want to make sure that it has been created successfully, we can simply write one line, similar to the one below:

```
print(os.access('some_new_project', os.F_OK))  # True
```

The directory `some_new_project` returned the Boolean `True` value. We specified a relative path, but you can indicate the absolute one, too.

To learn about access rights, `os.access()` uses ***uid*** (the identifier of a user on the system) or ***gid*** (the identifier of a user group). So, we can figure out the permissions of a certain category to directories or files. Here, the available modes are:

- `os.R_OK` to check for readability of the path (that is, the user has the permission to see the content of the file / directory);
- `os.W_OK` to check the writability of the path (permission to write to the file / directory);
- `os.X_OK` to check if the path contents can be executed.

In the example below, we specify the path to a plaintext file in `course`: and the result is a Boolean value. As you can see, the file is both writable and readable, but we cannot execute it, because it is a plaintext file.

```
print(os.access('course/list_of_students.txt', os.R_OK))  # True
print(os.access('course/list_of_students.txt', os.W_OK))  # True
print(os.access('course/list_of_students.txt', os.X_OK))  # False
```

§6. Removing directories and files

We will look at two functions for deleting directories and files.

- `os.remove()` deletes the specified *file*, the relative, or the full path to which we pass as an argument.

```
os.remove('course/course_plan.txt')

# checking for existence
os.access('course/course_plan.txt', os.F_OK)
# False
```

- `os.rmdir()` removes a single specified *directory*. Before using it, make sure that the directory you want to delete is empty. Otherwise, an `OSError` will be raised.

```
os.rmdir('course/students/year')

# checking for existence
os.access('course/students/year', os.F_OK)
# False
```

At this point, we will stop with the `os.` We will discuss the `os.path` functions in the two remaining sections.

§7. Path components in os.path

`os.path` is mainly used for manipulating paths. As we already mentioned, different operating systems use different conventions for pathnames, so there are actually several versions of this module with the very same functionality: for example, `posixpath` for UNIX-style paths or `ntpath` for Windows paths. We can always import `os` and then use `os.path` instead of working with them separately, the functions will work in the same way as in the modules suitable for a specific operating system.

We will start with four functions that allow us to manipulate the path components.

- `os.path.join()` joins several given components to create a new pathname. For example, we can pass it as an argument to the `os.makedirs()` to create nested directories. The example below shows the output for Linux first and then for Windows:

```
print(os.path.join('more_new_projects', 'more_new_plans'))
# more_new_projects/more_new_plans

print(os.path.join('more_new_projects', 'more_new_plans'))
# more_new_projects\\more_new_plans
```

- `os.path.split()` splits a pathname into a tuple ***(head, tail)***, where ***tail*** is the very last component of the given pathname and ***head*** is everything else preceding this last component. The example below illustrates it for the UNIX-style path and then for the Windows-style one:

```
print(os.path.split('/home/user/more_new_projects/more_new_plans'))
# ('/home/user/more_new_projects', 'more_new_plans')

print(os.path.split('C:\\Users\\User\\more_new_projects\\more_new_plans'))
# ('C:\\Users\\User\\more_new_projects', 'more_new_plans')
```

- Based on the latter function, the `os.path.dirname()` returns the directory of the path (*head*).

```
print(os.path.dirname('/home/user/more_new_projects/more_new_plans'))
# /home/user/more_new_projects

print(os.path.dirname('C:\\Users\\User\\more_new_projects\\more_new_plans'))
# C:\\Users\\User\\more_new_projects
```

- Accordingly, `os.path.basename()` returns the ***tail***, whether it is the file name or the name of another directory.

```
print(os.path.basename('/home/user/more_new_projects/more_new_plans'))
# more_new_plans

print(os.path.basename('C:\\Users\\User\\more_new_projects\\more_new_plans'))
# more_new_plans
```

§8. Path's validity

`os.path` also has several functions that allow us to check whether the given path is an absolute or a relative one; whether it refers to a file or directory. Below, we will list three examples of such functions.

- `os.path.isabs()` simply checks if the path we pass to it is the absolute one; in this case it returns True:

```
print(os.path.isabs('/home/user/more_new_projects/more_new_plans'))  # True

print(os.path.isabs('C:\\Users\\User\\more_new_projects\\more_new_plans'))  # True
```

- `os.path.isdir()` checks whether the given path refers to a directory.

```
print(os.path.isdir('/home/user/more_new_projects/more_new_plans'))  # True
```

- `os.path.isfile()`, on the contrary, checks whether the specified path refers to a file.

```
print(os.path.isfile('/home/user/more_new_projects/more_new_plans'))  # Fals
e
```

So far, in these two sections, we gave a small overview of the basic `os.path` functions.

§9. Summary

In this topic, we covered the functions of the `os` and `os.path` modules. The functionality of both is, of course, much larger and you can find the entire list of provided functions in the os module documentation and the os.path module documentation respectively.

Let's briefly sum up the functions we have discussed in the `os` module:

- `os.getcwd()` to learn the current working directory and `os.chdir()` to change it.
- `os.mkdir()` to create a single directory and `os.makedirs()` to create multiple nested folders.
- `os.listdir()` to get the listing of the directory's content and `os.rename()` to change the name of files and folders.
- `os.access()` to check the path for existence and determine what permissions to a directory or a file of a certain user or group is granted.
- `os.remove()` to remove a single file and `os.rmdir()` to delete a single empty directory.

We slightly covered `os.path` as well by glancing at the following functions:

- `os.path.join()` to construct the new pathname from the given components.
- `os.path.split()` to split the pathname and `os.path.dirname()` and `os.path.basename()` to return a certain part of it.
- `os.path.isabs()` to find out if the given path an absolute one and `os.path.isdir()` and `os.path.isfile()` to check if it is a directory or a folder.

Now, it's time to practice your new knowledge!

Report a typo

44 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

This content was created 3 months ago and updated 6 days ago. Share your feedback below in comments to help us improve it!

Comments (6) Hints (0) Useful links (1) Show discussion