Python Essentials 2:

Module 4

Miscellaneous

In this module, you will learn about:

- Generators, iterators and closures;
- Working with file-system, directory tree and files;
- Selected Python Standard Library modules (os, datetime, time, and calendar.)

Introduction to the os module

In this section, you'll learn about a module called os, which lets you interact with the operating system using Python.

It provides functions that are available on Unix and/or Windows systems. If you're familiar with the command console, you'll see that some functions give the same results as the commands available on the operating systems.

A good example of this is the mkdir function, which allows you to create a directory just like the mkdir command in Unix and Windows. If you don't know this command, don't worry.

You'll soon have the opportunity to learn the functions of the os module, to perform operations on files and directories along with the corresponding commands.

In addition to file and directory operations, the os module enables you to:

- get information about the operating system;
- manage processes;
- operate on I/O streams using file descriptors.

In a moment, you'll see how to get basic information about your operating system, although process management and working with file descriptors won't be discussed here, because these are more advanced topics that require knowledge of operating system mechanisms.

Ready?

Getting information about the operating system

Before you create your first directory structure, you'll see how you can get information about the current operating system. This is really easy because the os module provides a function called uname, which returns an object containing the following attributes:

- systemname — stores the name of the operating system;
- nodename — stores the machine name on the network;
- release — stores the operating system release;
- version — stores the operating system version;
- machine — stores the hardware identifier, e.g., x86_64.

Let's look at how it is in practice:

```
import os
print(os.uname())
```

Result:

```
posix.uname_result(sysname='Linux', nodename='192d19f04766', release='4.4.0-164-generic', version='#192-Ubuntu SMP Fri Sep 13 12:02:50 UTC 2019', machine='x86_64')
output
```

As you can see, the uname function returns an object containing information about the operating system. The above code was launched on Ubuntu 16.04.6 LTS, so don't be surprised if you get a different result, because it depends on your operating system.

Unfortunately, the uname function only works on some Unix systems. If you use Windows, you can use the uname function in the platform module, which returns a similar result.

The os module allows you to quickly distinguish the operating system using the name attribute, which supports one of the following names:

- posix — you'll get this name if you use Unix;
- nt — you'll get this name if you use Windows;
- java — you'll get this name if your code is written in Jython.

For Ubuntu 16.04.6 LTS, the name attribute returns the name posix:

```
import os
print(os.name)
```

Result:

```
posix
output
```

NOTE: On Unix systems, there's a command called uname that returns the same information (if you run it with the -a option) as the uname function.

Creating directories in Python

The os module provides a function called mkdir, which, like the mkdir command in Unix and Windows, allows you to create a directory. The mkdir function requires a path that can be relative or absolute. Let's recall what both paths look like in practice:

- my_first_directory — this is a relative path which will create the my_first_directory directory in the current working directory;
- ./my_first_directory — this is a relative path that explicitly points to the current working directory. It has the same effect as the path above;
- ../my_first_directory — this is a relative path that will create the my_first_directory directory in the parent directory of the current working directory;
- /python/my_first_directory — this is the absolute path that will create the my_first_directory directory, which in turn is in the python directory in the root directory.

Look at the code in the editor. It shows an example of how to create the my_first_directory directory using a relative path. This is the simplest variant of the relative path, which consists of passing only the directory name.

If you test your code here, it will output the newly created ['my_first_directory'] directory (and the entire content of the current working catalog).

The mkdir function creates a directory in the specified path. Note that running the program twice will raise a FileExistsError.

This means that we cannot create a directory if it already exists. In addition to the path argument, the mkdir function can optionally take the mode argument, which specifies directory permissions. However, on some systems, the mode argument is ignored.

To change the directory permissions, we recommend the chmod function, which works similarly to the chmod command on Unix systems. You can find more information about it in the documentation.

In the above example, another function provided by the os module named listdir is used. The listdir function returns a list containing the names of the files and directories that are in the path passed as an argument.

If no argument is passed to it, the current working directory will be used (as in the example above). It's important that the result of the listdir function omits the entries '.' and '..', which are displayed, e.g., when using the ls -a command on Unix systems.

NOTE: In both Windows and Unix, there's a command called mkdir, which requires a directory path. The equivalent of the above code that creates the my_first_directory directory is the mkdir my_first_directory command.

Recursive directory creation

The mkdir function is very useful, but what if you need to create another directory in the directory you've just created. Of course, you can go to the created directory and create another directory inside it, but fortunately the os module provides a function called makedirs, which makes this task easier.

The makedirs function enables recursive directory creation, which means that all directories in the path will be created. Let's look at the code in the editor and see how it is in practice.

```
import os

os.makedirs("my_first_directory/my_second_directory")
os.chdir("my_first_directory")
print(os.listdir())
```

The code should produce the following result:

```
['my_second_directory']
```

output

The code creates two directories. The first of them is created in the current working directory, while the second in the my_first_directory directory.

You don't have to go to the my_first_directory directory to create the my_second_directory directory, because the makedirs function does this for you. In the example above, we go to the my_first_directory directory to show that the makedirs command creates the my_second_directory subdirectory.

To move between directories, you can use a function called chdir, which changes the current working directory to the specified path. As an argument, it takes any relative or absolute path. In our example, we pass the first directory name to it.

NOTE: The equivalent of the makedirs function on Unix systems is the mkdir command with the -p flag, while in Windows, simply the mkdir command with the path:

- Unix-like systems:
  ```
  mkdir -p my_first_directory/my_second_directory
  ```

- Windows:
  ```
  mkdir my_first_directory/my_second_directory
  ```

Where am I now?

You already know how to create directories and how to move between them. Sometimes, when you have a really large directory structure that you navigate, you may not know which directory you're currently working in.

As you've probably guessed, the os module provides a function that returns information about the current working directory. It's called getcwd. Look at the code in the editor to see how to use it in practice.

```
import os

os.makedirs("my_first_directory/my_second_directory")
os.chdir("my_first_directory")
print(os.getcwd())
os.chdir("my_second_directory")
print(os.getcwd())
```

Result:

```
.../my_first_directory
.../my_first_directory/my_second_directory
```

output

In the example, we create the my_first_directory directory, and the my_second_directory directory inside it. In the next step, we change the current working directory to the my_first_directory directory, and then display the current working directory (first line of the result).

Next, we go to the my_second_directory directory and again display the current working directory (second line of the result). As you can see, the getcwd function returns the absolute path to the directories.

NOTE: On Unix-like systems, the equivalent of the getcwd function is the pwd command, which prints the name of the current working directory.

Deleting directories in Python

The os module also allows you to delete directories. It gives you the option of deleting a single directory or a directory with its subdirectories. To delete a single directory, you can use a function called rmdir, which takes the path as its argument. Look at the code in the editor.

The above example is really simple. First, the my_first_directory directory is created, and then it's removed using the rmdir function. The listdir function is used as proof that the directory has been removed successfully. In this case, it returns an empty list. When deleting a directory, make sure it exists and is empty, otherwise an exception will be raised.

To remove a directory and its subdirectories, you can use the removedirs function, which requires you to specify a path containing all directories that should be removed:

```
import os

os.makedirs("my_first_directory/my_second_directory")
os.removedirs("my_first_directory/my_second_directory")
print(os.listdir())
```

As with the rmdir function, if one of the directories doesn't exist or isn't empty, an exception will be raised.

NOTE: In both Windows and Unix, there's a command called rmdir, which, just like the rmdir function, removes directories. What's more, both systems have commands to delete a directory and its contents. In Unix, this is the rm command with the -r flag.

The system() function

All functions presented in this part of the course can be replaced by a function called system, which executes a command passed to it as a string.

The system function is available in both Windows and Unix. Depending on the system, it returns a different result.

In Windows, it returns the value returned by the shell after running the command given, while in Unix, it returns the exit status of the process.

Let's look at the code in the editor and see how it is in practice.

```
import os

returned_value = os.system("mkdir my_first_directory")
print(returned_value)
```

Result:

```
0
output
```

The above example will work in both Windows and Unix. In our case, we receive exit status 0, which indicates success on Unix systems.

This means that the my_first_directory directory has been created. As part of the exercise, try to list the contents of the directory where you created the my_first_directory directory.

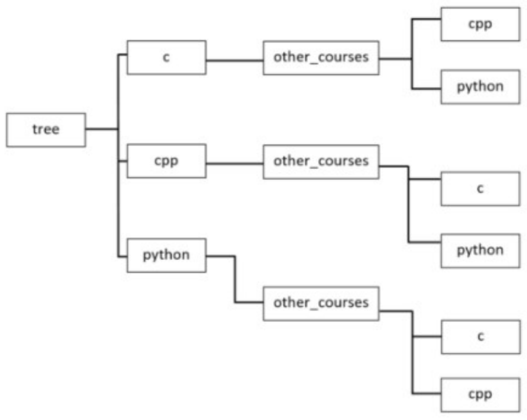
LAB

Estimated time
15-30 min

Level of difficulty
Easy

Objectives
improving the student's skills in interacting with the operating system;
practical use of known functions provided by the os module.
Scenario
It goes without saying that operating systems allow you to search for files and directories. While studying this part of the course, you learned about the functions of the os module, which have everything you need to write a program that will search for directories in a given location.

To make your task easier, we have prepared a test directory structure for you:



Your program should meet the following requirements:

Write a function or method called find that takes two arguments called path and dir. The path argument should accept a relative or absolute path to a directory where the search should start, while the dir argument should be the name of a directory that you want to find in the given path. Your program should display the absolute paths if it finds a directory with the given name.
The directory search should be done recursively. This means that the search should also include all subdirectories in the given path.
Example input:

path="./tree", dir="python"

Example output:

```
.../tree/python
.../tree/cpp/other_courses/python
.../tree/c/other_courses/python
```

Key takeaways

1. The uname function returns an object that contains information about the current operating system. The object has the following attributes:

- systemname (stores the name of the operating system)
- nodename (stores the machine name on the network)
- release (stores the operating system release)
- version (stores the operating system version)
- machine (stores the hardware identifier, e.g. x86_64.)

2. The name attribute available in the os module allows you to distinguish the operating system. It returns one of the following three values:

- posix (you'll get this name if you use Unix)
- nt (you'll get this name if you use Windows)
- java (you'll get this name if your code is written in something like Jython)

3. The mkdir function creates a directory in the path passed as its argument. The path can be either relative or absolute, e.g:

```
import os

os.mkdir("hello") # the relative path
os.mkdir("/home/python/hello") # the absolute path
```

Note: If the directory exists, a FileExistsError exception will be thrown. In addition to the mkdir function, the os module provides the makedirs function, which allows you to recursively create all directories in a path.

4. The result of the listdir() function is a list containing the names of the files and directories that are in the path passed as its argument.

It's important to remember that the listdir function omits the entries '.' and '..', which are displayed, for example, when using the ls -a command on Unix systems. If the path isn't passed, the result will be returned for the current working directory.

5. To move between directories, you can use a function called chdir(), which changes the current working directory to the specified path. As its argument, it takes any relative or absolute path.

If you want to find out what the current working directory is, you can use the getcwd() function, which returns the path to it.

6. To remove a directory, you can use the rmdir() function, but to remove a directory and its subdirectories, use the removedirs() function.

7. On both Unix and Windows, you can use the system function, which executes a command passed to it as a string, e.g.:

```
import os

returned_value = os.system("mkdir hello")
```

The system function on Windows returns the value returned by shell after running the command given, while on Unix it returns the exit status of the process.

Exercise 1

What is the output of the following snippet if you run it on Unix?

```
import os
print(os.name)
```

Check

```
posix
```

Exercise 2

What is the output of the following snippet?

```
import os

os.mkdir("hello")
print(os.listdir())
```

Check

```
['hello']
```