The *os* module

It lets you interact with the operating system.

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.

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

- Get information about the OS.
- Manage processes.
- Operate on IO streams using file descriptors.

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:

1. *systemname* — stores the name of the operating system.
2. *nodename* — stores the machine name on the network.
3. *release* — stores the operating system release.
4. *version* — stores the operating system version.
5. *machine* — stores the hardware identifier, e.g., x86_64.

This is how it's used.

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

```
posix.uname_result(sysname='Linux', nodename='a4789bc7ab70',
release='4.4.0-210-generic', version='#242-Ubuntu SMP Fri Apr 16 09:57:56
UTC 2021', machine='x86_64')
```

This is what I've got.

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 if your code is written in Jython.

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

This is what I've got.

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

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.

- $my\_first\_directory$ — 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.

The code here is 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.

```
import os
os.mkdir("my_first_directory")
print(os.listdir())
```

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, the $chmod$ function is recommended, which works similarly to the $chmod$ command on Unix systems. You can find more information about it in the documentation. (chmod man page (linuxcommand.org), chmod - Wikipedia)

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.

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.

Let's say that 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. Here's how it's used.

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

Expected output:

$$[\text{‘}my\_second\_directory\text{’}]$$

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.

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:

$$mkdir\ my\_first\_directory/my\_second\_directory$$

Getting current directory

The os module provides a function that returns information about the current working directory. It's called $getcwd$.

```
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())
```

Here’s how it’s used.

Expected output:

$$\ldots/my\_first\_directory$$

$$\ldots/my\_first\_directory/my\_second\_directory$$

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.

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

Deleting directory

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.

```python
import os
os.mkdir("my_first_directory")
print(os.listdir())
os.rmdir("my_first_directory")
print(os.listdir())
```

Here is how it's used.

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:

```python
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.

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.

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

Expected output:

```
0
```

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.