**The**  **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 function, which allows you to create a directory just like the command in Unix and Windows.

In addition to file and directory operations, the 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 module provides a function called , which returns an object containing the following attributes:

1. — stores the name of the operating system.
2. — stores the machine name on the network.
3. — stores the operating system release.
4. — stores the operating system version.
5. — 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 function only works on some Unix systems. If you use Windows, you can use the function in the platform module, which returns a similar result.

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

* — you’ll get this name if you use Unix.
* — you’ll get this name if you use Windows.
* — 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 that returns the same information (if you run it with the -a option) as the function.

**Creating directories**

The module provides a function called , which, like the command in Unix and Windows, allows you to create a directory. The function requires a path that can be relative or absolute.

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

The code here is an example of how to create the 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 directory (and the entire content of the current working catalog).

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

This means that we cannot create a directory if it already exists. In addition to the path argument, the 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 function is recommended, which works similarly to the command on Unix systems. You can find more information about it in the documentation. ([chmod man page (linuxcommand.org)](https://linuxcommand.org/lc3_man_pages/chmod1.html), [chmod - Wikipedia](https://en.wikipedia.org/wiki/Chmod))

In the above example, another function provided by the module named is used. The 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 function omits the entries '.' and '..', which are displayed, e.g., when using the command on Unix systems.

In both Windows and Unix, there's a command called , which requires a directory path. The equivalent of the above code that creates the directory is the 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 module provides a function called , which makes this task easier.

The 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:

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

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

To move between directories, you can use a function called , 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 function on Unix systems is the command with the flag, while in Windows, simply the command with the path:

Unix-like systems:

Windows:

**Getting current directory**

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

*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:

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

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

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

**Deleting directory**

The 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 , which takes the path as its argument.

*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 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 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 , which, just like the function, removes directories. What's more, both systems have commands to delete a directory and its contents. In Unix, this is the command with the flag.

**The**  **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 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.

*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 directory has been created. As part of the exercise, try to list the contents of the directory where you created the directory.