Directory and file functions in Python

os module in python provides several methods to work with directories and files. We will look at some of those functions in this lesson

os.chdir() and os.getcwd()

The **os.chdir** function allows us to change the directory that we're currently running our Python session in. If you want to actually know what path you are currently in, then you would call **os.getcwd()**. Let's try them both out:

```
import os

print(os.getcwd())
# '/usercode'

os.chdir('/var')
print(os.getcwd())
# '/var'
```

The code above shows us that we started out in the Python directory by default when we run this code in IDLE. Then we change folders using **os.chdir()**. Finally we call os.getcwd() a second time to make sure that we changed to the folder successfully.

os.mkdir() and os.makedirs()

You might have guessed this already, but the two methods covered in this section are used for creating directories. The first one is **os.mkdir()**, which allows us to create a single folder. Let's try it out:

```
import os
os.mkdir("test")

path = r'/usercode/pytest'
os.mkdir(path)
```

The first line of code will create a folder named **test** in the current directory. You can use the methods in the previous section to figure out where you just ran your code if you've forgotten. The second example assigns a path to a variable and then we pass the path to os.mkdir(). This allows you to create a folder anywhere on your system that you have permission to.

The **os.makedirs()** function will create all the intermediate folders in a path if they don't already exist. Basically this means that you can created a path that has nested folders in it. I find myself doing this a lot when I create a log file that is in a dated folder structure, like Year/Month/Day. Let's look at an example:

```
import os

path = r'/usercode/pytest\2017\02\01'
os.makedirs(path)
```

What happened here? This code just created a bunch of folders! If you still had the **pytest** folder in your system, then it just added a **2014** folder with another folder inside of it which also contained a folder. Try it out for yourself using a valid path on your system.

os.remove() and os.rmdir()

The **os.remove()** and **os.rmdir()** functions are used for deleting files and directories respectively. Let's look at an example of **os.remove()**:

```
import os
os.remove("test.txt")
```

This code snippet will attempt to remove a file named **test.txt** from your current working directory. If it cannot find the file, you will likely receive some sort of error. You will also receive an error if the file is in use (i.e. locked) or you don't have permission to delete the file. You might also want to check out **os.unlink**, which does the same thing. The term **unlink** is the traditional Unix name for this procedure.

Now let's look at an example of **os.rmdir()**:

```
import os
os.rmdir("pytest")
```

The code above will attempt to remove a directory named **pytest** from your current working directory. If it's successful, you will see that the directory no longer exists. An error will be raised if the directory does not exist, you do not have permission to remove it or if the directory is not empty. You might also want to take a look at **os.removedirs()** which can remove nested empty directories recursively.

os.rename(src, dst)

The **os.rename()** function will rename a file or folder. Let's take a look at an example where we rename a file:

```
import os
os.rename("test.txt", "pytest.txt")
```

In this example, we tell **os.rename** to rename a file named **test.txt** to **pytest.txt**. This occurs in our current working directory. You will see an error occur if you try to rename a file that doesn't exist or that you don't have the proper permission to rename the file.

There is also an **os.renames** function that recursively renames a directory or file.

os.startfile()

The **os.startfile()** method allows us to "start" a file with its associated program. In other words, we can open a file with it's associated program, just like when you double-click a PDF and it opens in Adobe Reader. Let's give it a try!

```
import os
os.startfile(r'/users/mike/Documents/labels.pdf')
```

In the example above, I pass a fully qualified path to **os.startfile** that tells it to open a file called **labels.pdf**. On my machine, this will open the PDF in Adobe Reader. You should try opening your own PDFs, MP3s, and photos using this method to see how it works.

os.walk()

The **os.walk()** method gives us a way to iterate over a root level path. What this means is that we can pass a path to this function and get access to all its sub-directories and files. Let's use one of the Python folders that we have handy to test this function with. We'll use: C:\Python27\Tools

```
import os
path = r'/usr/bin'
for root, dirs, files in os.walk(path):
        print(root)
```

If you want, you can also loop over **dirs** and **files** too. Here's one way to do it:

```
import os
```

```
path = r'/usr/bin'
for root, dirs, files in os.walk(path):
        print(root)

        for _dir in dirs:
            print(_dir)
        for _file in files:
            print(_file)
```

This piece of code will print a lot of stuff out, so I won’t be showing its output here, but feel free to give it a try. Now we’re ready to learn about working with paths!