## PYTHON INTEGRATION

PHP and Python are two of the most popular and influential programming languages. They both have their strengths and weaknesses and share many standard software development capabilities. Although Rasmus Lerdorf developed PHP as a web language, and Guido van Rossum created Python as a general-purpose programming language. Assessing each language and comparing their strengths across several core elements can help you make the correct choice.

Development Environment (PHP < Python)
Language Complexity (PHP > Python)
Extendibility (PHP > Python)
Security (PHP > Python)
Scalability and Adaptability (=)
Documentation and Community Support(=)

## EXECUTING PYTHON SCRIPT USING PHP:

To run a Python script from PHP, we can use the shell_exec function.
**Step :1** create python file “x.py” in “www\wamp”

```
Print(“hello”)
```

**Step:2** create php file for calling(execute) python script.

```
<?PHP
$command_exec = escapeshellcmd('python x.py');
$str_output = shell_exec($command_exec);
echo $str_output;
 ?>
```

To call escapeshellcmd to escape the command string.Then we call shell_exec to run the $command.
And we get the output from $output

## ESCAPESHELLCMD() METHOD:

**escapeshellcmd()** escapes any characters in a string that might be used to trick a shell command into executing arbitrary commands. This function should be used to make sure that any data coming from user input is escaped before this data is passed to the exec() or system() functions, or to the backtick operator.
The command line is a dangerous place for unescaped characters. Never pass unmodified user input to one of PHP's shell-execution functions. Always escape the appropriate characters in the command and the arguments. Following characters are preceded by a backslash: &#;`|*?~<>^()[]{}$\, \x0A and \xFF. ' and " are escaped only if they are not paired. On Windows, all these characters plus % and ! are preceded by a caret (^).
Syntax:

```
escapeshellcmd ( string $command )
```

Example:

```
<?php
$command = "\\%!** Hello  World";
$escaped_command = escapeshellcmd($command);
 echo ($escaped_command);
?>
```

Output

!Hello World

## SHELL_EXEC() METHOD :

used to execute the commands via shell and return the complete output as a string. The shell_exec is an alias for the backtick operator

syntax:

```
shell_exec( $cmd )
```

example: display list of file and directory

```
$output = shell_exec(‘dir');
echo "$output";
```

## EXEC() METHOD:

The exec()is used to execute an external program and returns the last line of the output. It also returns NULL if no command run properly.

Syntax:

```
exec( $command, $output)
```

example: display last line of output

```
exec("dir",$output1);
echo $output1;
```

## SYSTEM() :

it is for executing a system command and immediately displaying the output - presumably text.

Example:

```
system("dir",$output1);
echo $output1;
```

## THE OS MODULE

The **OS** **module** in **Python** provides functions for interacting with the **operating system**. **OS** comes under **Python's** standard utility **modules**. we will be covering the following:

- os.chdir() os.getcwd()
- os.mkdir() os.makedirs()
- os.remove() os.rename()
- os.rmdir() os.walk()
- os.path write()
- read() close()

**firstly “import** os” for executing all commands under os module.

## 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().**

```
>>> os.getcwd()
'C:\\Python27'
>>> os.chdir("c:\Users\mike\Documents")
>>> os.getcwd()
'c:\\Users\\mike\\Documents'
```

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()

used for creating directories. The first one is **os.mkdir()**, which allows us to create a single folder. Let's try it out:

```
>>> os.mkdir("test")
>>> path = 'C:\wamp\www\test\pytest'
>>> os.mkdir(path)
```

The first line of code will create a folder named **test** in the current directory.

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:

```
>>> path = ' C:\wamp\www\test\pytest\2014\02\19'
>>> os.makedirs(path)
```

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()**:

```
>>> 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.

an example of **os.rmdir()**:

```
>>> 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:

```
>>> 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.

## 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

```
>>> path = 'C:\Python27\Tools'
>>> for root, dirs, files in os.walk(path):
        print(root)
```

```
C:\Python27\Tools
C:\Python27\Tools\i18n
C:\Python27\Tools\pynche
C:\Python27\Tools\pynche\X
C:\Python27\Tools\Scripts
C:\Python27\Tools\versioncheck
C:\Python27\Tools\webchecker
```

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

```
>>> 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!

## OS.PATH

The **os.path** sub-module of the **os** module has lots of great functionality built into it. We'll be looking at the following functions:

- basename
- dirname
- exists
- isdir and isfile
- join
- split

There are lots of other functions in this sub-module.

## OS.PATH.BASENAME

The **basename** function will return just the filename of a path. Here is an example:

```
>>> os.path.basename(' C:\wamp\www\test\pytest \p1.py')
'p1.py'
```

I have found this useful whenever I need to use a filename for naming some related file, such as a log file. This happens a lot when I'm processing a data file.

## OS.PATH.DIRNAME

The **dirname** function will return just the directory portion of the path. It's easier to understand if we take a look at some code:

```
>>> os.path.dirname(' C:\wamp\www\test\pytest\p1.py')
' C:\wamp\www\test\pytest '
```

In this example, we just get the directory path back. This is also useful when you want to store other files next to the file you're processing, like the aforementioned log file.

## OS.PATH.EXISTS

The **exists** function will tell you if a path exists or not. All you have to do is pass it a path. Let's take a look:

```
>>> os.path.exists(' C:\wamp\www\test\pytest\p1.py')
True
>>> os.path.exists(' C:\wamp\www\test\pytest\fake.py')
False
```

In the first example, we pass the **exists** function a real path and it returns **True**, which means that the path exists. In the second example, we passed it a bad path and it told us that the path did not exist by returning **False**.

## OS.PATH.ISDIR / OS.PATH.ISFILE

The **isdir** and **isfile** methods are closely related to the **exists** method in that they also test for existence. However, **isdir** only checks if the path is a directory and **isfile** only checks if the path is a file. If you want to check if a path exists regardless of whether it is a file or a directory, then you'll want to use the **exists** method.

Examples:

```
>>> os.path.isfile(' C:\wamp\www\test\pytest\p1.py')
True
>>> os.path.isdir(' C:\wamp\www\test\pytest\p1.py')
False
>>> os.path.isdir(' C:\wamp\www\test\pytest ')
True
>>> os.path.isfile(' C:\wamp\www\test\pytest1')
False
```

## OS.LISTDIR

This function just lists out the files and directories present in the current working directory.

```
>>> import os
>>> os.listdir()
```

## OS.PATH.JOIN

The **join** method give you the ability to join one or more path components together using the appropriate separator. For example, on Windows, the separator is the backslash, but on Linux, the separator is the forward slash. Here's how it works:

```
>>> os.path.join(' C:\wamp\www\test\pytest ', 'p1.py')
' C:\wamp\www\test\pytest \p1.py'
```

In this example, we joined a directory path and a file path together to get a fully qualified path.

## OS.PATH.SPLIT

The **split** method will split a path into a tuple that contains the directory and the file.

```
>>> os.path.split(' C:\wamp\www\test\pytest \p1.py')
(' C:\wamp\www\test\pytest ', 'p1.py')
```

This example shows what happens when we path in a path with a file. Let's see what happens if the path doesn't have a filename on the end:

```
 >>> os.path.split(' C:\wamp\www\test\pytest ')
(' C:\wamp\www\test\pytest ', 'p1')
```

As you can see, it took the path and split it in such a way that the last sub-folder became the second element of the tuple with the rest of the path in the first element.

For our final example, I thought you might like to see a commmon use case of the **split**:

```
>>> dirname, fname = os.path.split(' C:\wamp\www\test\pytest\p1.py')
>>> dirname
' C:\wamp\www\test\pytest '
>>> fname
'p1.py'
```

This shows how to do multiple assignment. When you split the path, it returns a two-element tuple. Since we have two variables on the left, the first element of the tuple is assigned to the first variable and the second element to the second variable.

## OS.OPEN()

Python method open() opens the file file and set various flags according to flags and possibly its mode according to mode.The default mode is 0777 (octal), and the current umask value is first masked out.

### SYNTAX:

```
os.open(file, flags[, mode]);
```

### PARAMETERS

file − File name to be opened.

flags − The following constants are options for the flags. They can be combined using the bitwise OR operator |. Some of them are not available on all platforms.

- os.O_RDONLY − open for reading only
- os.O_WRONLY − open for writing only
- os.O_RDWR − open for reading and writing
- os.O_NONBLOCK − do not block on open
- os.O_APPEND − append on each write
- os.O_CREAT − create file if it does not exist
- os.O_TRUNC − truncate size to 0
- os.O_EXCL − error if create and file exists
- os.O_SHLOCK − atomically obtain a shared lock
- os.O_EXLOCK − atomically obtain an exclusive lock
- os.O_DIRECT − eliminate or reduce cache effects
- os.O_FSYNC − synchronous writes
- os.O_NOFOLLOW − do not follow symlinks

mode − This work in similar way as it works for chmod()

## OS. READ():

os.read() method in Python is used to read at most n bytes from the file associated with the given file descriptor.

### SYNTAX:

```
os.read(fd, n)
```

### PARAMETER:

fd: A file descriptor representing the file to be read.
n: An integer value denoting the number of bytes to be read from the file associated with the given file descriptor fd.

## OS.CLOSE()

os.close() method in Python is used to close the given file descriptor, so that it no longer refers to any file or other resource and may be reused.

### SYNTAX:

```
os.close(fd)
```

## PARAMETER:

fd: A file descriptor, which is to be closed.

Example of (open,read and close)

```
import os

# Open the file and get # the file descriptor associated # with it using os.open() method
fd = os.open(“data.csv”, os.O_RDONLY)

# Number of bytes to be read
n = 50

# Read at most n bytes  # from file descriptor fd # using os.read() method
readBytes = os.read(fd, n)

# Print the bytes read
print(readBytes)

# close the file descriptor
os.close(fd)
```

## OS.WRITE()

os.write() method in Python is used to write a bytestring to the given file descriptor.

## SYNTAX:

os.write(fd, str)

## PARAMETER:

fd: The file descriptor representing the target file.
str: A bytes-like object to be written in the file.

Example:

```
import os
 # Open the file and get# the file descriptor associated# with it using os.open() method
fd = os.open("x.txt", os.O_RDWR|os.O_CREAT )
 # String to be written
s = "I like apple"
 # Convert the string to bytes
line = str.encode(s)

# Write the bytestring to the file # associated with the file # descriptor fd and get the number of
# Bytes actually written
numBytes = os.write(fd, line)
print("Number of bytes written:", numBytes)
# close the file descriptor
os.close(fd)
```

## SUBPROCESS MODULE IN PYTHON:

Subprocess in Python is a module used to run new codes and applications by creating new processes. It lets you start new applications right from the Python program you are currently writing. So, if you want to run external programs from a git repository or codes from C or C++ programs, you can use subprocess in Python.

The subprocess Module

The **subprocess** module gives the developer the ability to start processes or programs from Python. In other words, you can start applications and pass arguments to them using the subprocess module. The subprocess module was added way back in Python 2.4 to replace the **os** modules set of os.popen, os.spawn and os.system calls as well as replace popen2 and the old **commands** module. We will be looking at the following aspects of the subprocess module:

the call function

the Popen class

how to communicate with a spawned process

### THE CALL FUNCTION

The subprocess module provides a function named **call**. This function allows you to call another program, wait for the command to complete and then return the return code. It accepts one or more arguments as well as the following keyword arguments (with their defaults): stdin=None, stdout=None, stderr=None, shell=False.

Let’s look at a simple example:

```
>>> import subprocess
>>> subprocess.call("notepad.exe")
0
```

If you run this on a Windows machine, you should see Notepad open up. You will notice that IDLE waits for you to close Notepad and then it returns a code zero (0). This means that it completed successfully. If you receive anything except for a zero, then it usually means you have had some kind of error.

### THE POPEN CLASS

The **Popen** class executes a child program in a new process. Unlike the **call** method, it does not wait for the called process to end unless you tell it to using by using the **wait** method. It is like os.system command. Python method **popen()** opens a pipe to or from command.The return value is an open file object connected to the pipe, which can be read or written depending on whether mode is 'r' (default) or 'w'.

#### SYNTAX:

```
os.popen(command[, mode])
```

#### EXAMPLE

```
>>> program = "notepad.exe"
>>> subprocess.Popen(program)
```

#### EXAMPLE:

```
import subprocess
cm='dir'

p1= subprocess.Popen(cm,shell=True)
p1.wait()
if p1.returncode==0:
    print("sucessful run")
else:
    print(p1.stderr)
```

## EXPLANATION:

- Import subprocess module.
- Run 'dir' command and wait for process completion.
- If process completed successful it's return 0. Otherwise 1(none).
- If no error found display "successfully run" otherwise display error using stderr.

## SOME OTHER PARAMETERS OF POPEN:

```
#display output on terminal
        p1= subprocess.Popen(cm,shell=True)
        print(p1)
#display output store in p1 variable using subprocess.pipe, print on terminal using print(p1)
        p1= subprocess.Popen(cm,stdout=subprocess.PIPE,shell=True)
#TO PRINT RETURNCODE
        print(p1.returncode)
#PRINT ARGUMENTS
        print(p1.args)
#not display output and pass to DEVNULL file using subprocess.DEVNULL
        p1= subprocess.Popen(cm,stdout=subprocess.DEVNULL,shell=True)
```

Note : that using the **wait** method can cause the child process to deadlock when using the stdout/stderr=PIPE commands when the process generates enough output to block the pipe. You can use the **communicate** method to alleviate this situation.

## COMMUNICATE

There are several ways to communicate with the process you have invoked. to use the subprocess module's **communicate** method.

## SYNTAX:

Variableofincommingprocess.communicate()

## EXAMPLE:

```
Import subprocess
args = 'dir'
```

```
p1=
subprocess.Popen(args,stdout=subprocess.PIPE,stderr=subprocess.PIPE,shell=True,universal_n
ewlines=True)
o,e=p1.communicate()
print('out',format(o))
print('error',format(e))
if p1.returncode==0:
   print("sucessful run")
else:
   print(p1.stderr)
```

## EXPLANATION:

it takes output of first process(p1) for further process using p1.communicate().
We can store value of stdout and stderr in variable 'o' and 'e', because it will pass using stdout=subprocess.PIPE and stderr=subprocess.PIPE respectively.
Output will display using print()

In this code example, we create an **args** variable to hold our list of arguments. Then we redirect standard out (stdout) to our subprocess so we can communicate with it. The **communicate** method itself allows us to communicate with the process we just spawned. We can actually pass input to the process using this method. But in this example, we just use communicate to read from standard out. You will notice when you run this code that communicate will wait for the process to finish and then returns a two-element tuple that contains what was in stdout and stderr. That last line that says "None" is the result of stderr, which means that there were no errors.

## CHECK-CALL():

to run new applications or programs through Python code by creating new processes. It also helps to obtain the input/output/error pipes as well as the exit codes of various commands. check_call() **returns as soon as /bin/sh process exits without waiting for descendant processes** (assuming shell=True as in your case). check_output() waits until all output is read.

**subprocess.check_call(args, *, stdin=None, stdout=None, stderr=None, shell=False)**

**Parameters:**
args=The command to be executed.Several commands can be passed as a string by separated by ";".
stdin=Value of standard input stream to be passed as (os.pipe()).
stdout=Value of output obtained from standard output stream.
stderr=Value of error obtained(if any) from standard error stream.
shell=Boolean parameter.If True the commands get executed through a new shell environment.

## EXAMPLE:

```
cm=['echo','hello']
p1=subprocess.check_call(cm,shell=True,universal_newlines=True)
```

print(p1)

## CHECK-OUTPUT():

check_output() is **used to get the output of the calling program in python**. It has 5 arguments; args, stdin, stderr, shell, universal_newlines. The args argument holds the commands that are to be passed as a string.

**check_output(args, *, stdin=None, stderr=None, shell=False, universal_newlines=False)**
**Parameters:**
args=The command to be executed. Several commands can be passed as a string by separated by ";".
stdin=Value of standard input stream to be passed as pipe(os.pipe()).
stdout=Value of output obtained from standard output stream.
stderr=Value of error obtained(if any) from standard error stream.
shell=boolean parameter.If True the commands get executed through a new shell environment.
universal_newlines=Boolean parameter.If true files containing stdout and stderr are opened in universal newline mode.

EXAMPLE:

```
Import subprocess
cm='dir'
p1=subprocess.check_output(cm,shell=True,universal_newlines=True)
print(p1)
```

## DECODE():

This method is used to convert from one encoding scheme, in which argument string is encoded to the desired encoding scheme. This works opposite to the encode. It accepts the encoding of the encoding string to decode it and returns the original string.
*Syntax :*
    ***decode(encoding, error)***
**Parameters:**
**encoding :** Specifies the encoding on the basis of which decoding has to be performed.
**error :** Decides how to handle the errors if they occur, e.g 'strict' raises Unicode error in case of exception and 'ignore' ignores the errors occurred. decode with error replace implements the 'replace' error handling.

EXAMPLE:

```
a = 'This is a bit möre cömplex sentence.'
print('Original string:', a)
 # Encoding in UTF-8
encoded_bytes = a.encode('utf-8', 'replace')
# Trying to decode via ASCII, which is incorrect
decoded_correct = encoded_bytes.decode('utf-8', 'replace')
print('Correctly Decoded string:', decoded_correct)
```

## EXECUTE PHP FILE IN PYTHON

step;1 create p1.php

<?php

Echo “hello”;

?>

Step:2 create python file at same path.

```
import subprocess
proc = subprocess.Popen("php c:/wamp/www/test/p1.php", shell=True,
stdout=subprocess.PIPE)
script_response = proc.stdout.read()
print(script_response)
```

Note if not error like ->> set environmental variable for php