Python's sys module

Learn the usage of sys module in python

WE'LL COVER THE FOLLOWING

The **sys** module provides system specific parameters and functions. We will be narrowing our study down to the following:

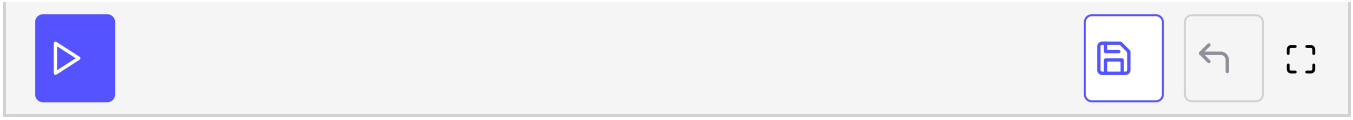
- sys.argv
- sys.executable
- sys.exit
- sys.modules
- sys.path
- sys.platform
- sys.stdin/stdout/stderr

sys.argv

The value of **sys.argv** is a Python list of command line arguments that were passed to the Python script. The first argument, **argv[0]** is the name of the Python script itself. Depending on the platform that you are running on, the first argument may contain the full path to the script or just the file name. You should study the documentation for additional details

Let's try out a few examples to familiarize ourselves with this little tool:

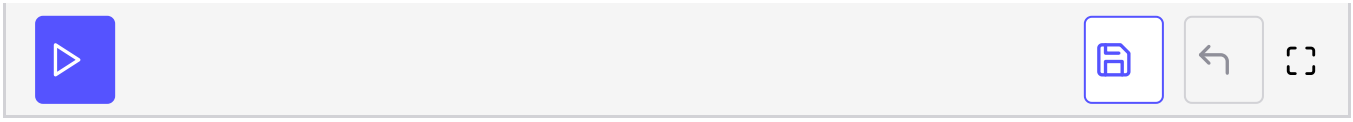
```
import sys
print(sys.argv)
```



sys.executable

The value of **sys.executable** is the absolute path to the Python interpreter. This is useful when you are using someone else's machine and need to know where Python is installed. On some systems, this command will fail and it will return an empty string or None. Here's how to use it:

```
import sys
print(sys.executable)
```



sys.exit

The **sys.exit()** function allows the developer to exit from Python. The **exit** function takes an optional argument, typically an integer, that gives an exit status. Zero is considered a "successful termination". Be sure to check if your operating system has any special meanings for its exit statuses so that you can follow them in your own application. Note that when you call **exit**, it will raise the **SystemExit** exception, which allows cleanup functions to work in the **finally** clauses of **try** / **except** blocks.

Let's take a look at how to call this:

```
import sys
sys.exit(0)
```

When you run this code in IDLE, you will see the SystemExit error raised. Let's create a couple of scripts to test this out. First you'll want to create a master

script, a program that will call another Python script. Let's name it "call_exit.py". Put the following code into it:

```
# call_exit.py
import subprocess

code = subprocess.call(["python.exe", "exit.py"])
print(code)
```

Now create another Python script called "exit.py" and save it in the same folder. Put the following code into it:

```
import sys

sys.exit(0)
```

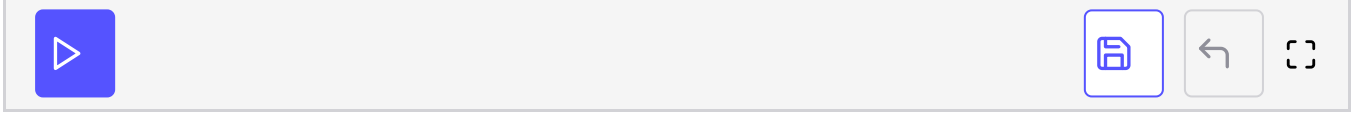
Now let's try running this code:

You can see that the exit script we wrote returned a zero, so it ran successfully. You have also learned how to call another Python script from within Python!

sys.path

The sys module's **path** value is a list of strings that specifies the search path for modules. Basically this tells Python what locations to look in when it tries to import a module. According to the Python documentation, **sys.path** is initialized from an environment variable called PYTHONPATH, plus an installation-dependent default. Let's give it a try:

```
import sys
print(sys.path)
```



This can be very useful for debugging why a module isn't getting imported. You can also modify the path. Because it's a list, we can add or delete paths from it. Here's how to add a path:

```
sys.path.append("/path/to/my/module")
```

I'll leave deleting a path as an exercise for the reader.

sys.platform

The **sys.platform** value is a platform identifier. You can use this to append platform specific modules to **sys.path**, import different modules depending on platform or run different pieces of code. Let's take a look:

```
import sys
print(sys.platform)
```

This tells us that Python is running on a Windows machine. Here's an example of how we might use this information:

```
import sys
os = sys.platform
if os == "win32":
    # use Window-related code here
    import _winreg
elif os.startswith('linux'):
    # do something Linux specific
    import subprocess
    subprocess.Popen(["ls", "-l"])
```

The code above shows how we might check to see if we're using a particular operating system. If we're on Windows, we'll get some information from the Window's Registry using a Python module called **_winreg**. If we're on Linux, we might execute the **ls** command to get information about the directory we're in.

sys.stdin / stdout / stderr

The **stdin**, **stdout** and **stderr** map to file objects that correspond to the interpreter's standard input, output and error streams, respectively. **stdin** is used for all input given to the interpreter except for scripts whereas **stdout** is used for the output of **print** and **expression** statements. The primary reason I mention this is that you will sometimes need to redirect stdout or stderr or

both to a file, such as a log or to some kind of display in a custom GUI you have created. You could also redirect stdin, but I have rarely seen this done.

Wrapping Up

There are many other values and methods in the **sys** module. Be sure to look it up in the Python documentation, section 27.1. You have learned a lot in this chapter. You now know how to exit a Python program, how to get platform information, working with arguments passed on the command line and much more. In the next chapter, we'll be learning about Python threads!