Shell Scripting

Shell Gathering, early 19th Century
葛飾 北斎 Katsushika Hokusai (1760-1849)

Learning Objectives

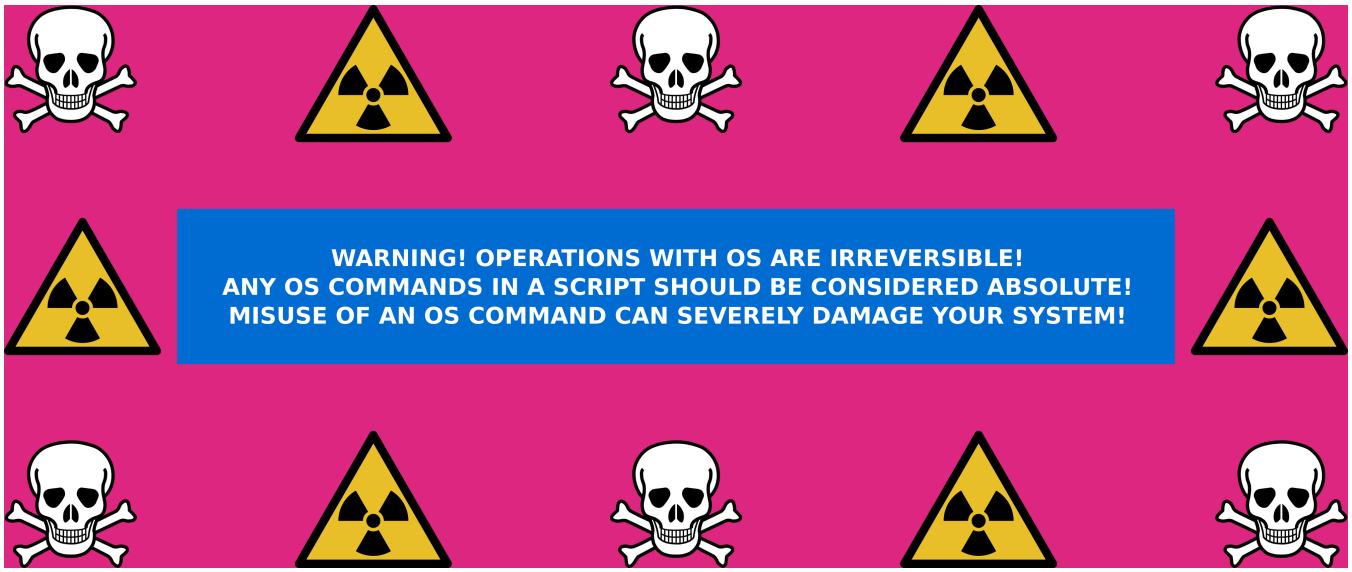
- Understand that the shell allows us to interact with the operating system in fundamental or near-fundamental ways
- Consider how Python might be used to automate portions of our workflow

Japanese cuisine (和食 "washoku") is fundamentally linked to the sea. Any number of dishes rely on the various fresh- and saltwater fishes, mollusks, and crustaceans of the expanse. One class of dishes, shellfish (貝 "kai"), consists of savory innards surrounded by a hard calcium shell.

Like the shell of shellfish, the **shell** of an operating system protects the soft, tasty innards. In this case, however, the "soft tasty innards" are actually very sensitive system components which are exceedingly unforgiving in preventing you from making a mistake. When used properly, we can use Python to interact with the operating system, giving us nearly complete control of system execution. This is often done using the **os module**.

Any number of fundamental system operations, such as file/folder creation, reading, updating, and deleting (CRUDing), can be performed using this module. os is often used in highly automated workflows, allowing the script to handle temporary files and enforce naming conventions for inputs and outputs, among other standardization measures. As well, certain resource requests (such as specific amounts of memory) can be handled using this module.

With great power comes great responsibility. *Essentially none of the operations performed with os are reversible.* We should repeat:



DO NOT UNDERESTIMATE THE POWER OF OS! As opposed to our normal mode of interaction with the operating system (the GUI), os will **NOT** ask our permission before CRUDing. Therefore, you should consider every command absolute, and, preferably, run any script which requires os in a sandbox (contained) environment (usually a dedicated folder or virtual machine which has a similar structure to the master machine on which the script is to be used).

Figure 1: A Hokusai metaphor for what happens when you run the wrong os command.

OK, enough of the scary stuff. We can first ask ourselves what a shell is by asking ourselves what a terminal is.

A shell is a terminal by any other name...

A long time ago, before computers were the size of frying pans, there were terminals. As in, a physical place where you would access a computer (or mainframe as they were often called). Terminals were/are also called consoles. But the shell, which is contained in the terminal, is what actually sits on top of the **kernel**. The kernel is the most fundamental part of an operating system and essentially *is* the operating system. You will often here the terms **shell** and **terminal** used interchangeably, but these are actually distinct concepts. To add to the confusion, the shell is often described as consisting of a **scripting language** (bash most often in the case of Linux). For this reason, anything having to do with using commands in the terminal via the shell is refered to as **shell scripting**.

Confused yet?[1] Don't worry, the main thing to remember is that the shell is essentially what stands between us and the kernel. This is for good reason—do you think you'd be very happy if you were given direct access to your hard drive and you accidentally erased half of it by using the wrong command line argument? The shell is there to prevent such catastrophic commands from being run. But, you can also negotiate a bit with it at the times you do need such power.

I have seen *.sh scripts do "command line stuff", why don't we just use shell scripting for everything?

Shell scripting using bash is very, very powerful. But it is also very, very insufferable when trying to do things in a standardized, yet flexible way.

Here is an example of a shell script[2]:

```
# ...

# USAGE
#   yes_or_no "title" "text" height width ["yes text"] ["no text"]
# INPUTS
#   $LINE = (y/n)
#   $_DEFAULT = (optional)
yes_or_no() {
        if [ "$LINE" == "y" ]; then
                echo -e "\e[1m$1\e[0m"
                echo '$2' | fold -w $4 -s
                while read -e -n 1 -i "$_DEFAULT" -p "Y for ${5:-Yes}, \
                N for ${6:-No}[Y/N]" _yesno; do
                        case $_yesno in
                        [yY]|1)
                                return 0
                                ;;
```

[1] See https://www.geeksforgeeks.org/difference-between-terminal-console-shell-and-command-line/ for a more nuanced explanation of these concepts.

[2] This is a modified version of the script found at https://en.wikibooks.org/wiki/Bash_Shell_Scripting#The_read_built-in

```
                        [nN]|0)
                                return 1
                                ;;
                        *)
                                echo -e "\e[1;31mINVALID INPUT\x21\e[0m"
                esac
        else whiptail --title "${1:-Huh?}" \
         --yesno "${2:-Are you sure?}" ${3:-10} ${4:-80}\
        --yes-button "${5:-Yes}" --no-button "$6{:-No}"; return $?
        fi
}

# ...
```

This code, to put it lightly, is ugly. The variables, loops, and if/else logic are hard to follow, and even harder to modify. This script may well do its job, but if another person were to come along, it would not be so easy for them to make something new out of this.

There *are* times when a simple shell script will suffice for the need at hand (for example, a quick re-naming of files in a folder). However, any more complex logic starts to get into muddy waters, and it is best to resort to Python for handling such situations.

Nearly anything you can do in the shell you can do in Python. Even better, you can do things in Python that are much harder to deal with in the shell (like making API requests). It is for this reason that Python is often considered the language of choice for automating pipelines. If you're ever in a situation where you need to CRUD a lot of folders/files, then your first instinct should be to set up the core logic in Python.

The os module in Python: the basics

The operations of os are best given in summary form. Consider Table 1[3]. Within the table, we can see there are benign commands [such as os.getcwd()], and there are not so benign commands [such as os.rmdir()]. **IT IS IMPERATIVE THAT YOU DO NOT USE THE WRONG COMMAND IN YOUR SCRIPT!!!**

[3] Adapted from `https://www.geeksforgeeks.org/os-module-python-examples/`

Command (dangerous)	**Purpose**
os.getcwd()	Get the current working directory. The folder where the Python script is running is known as the Current Working Directory (CWD). This may be a different directory than where the Python script is located.
os.chdir(*path*)	Change the CWD to *path*.
os.mkdir(*path*, *mode*)	Create the directory at *path* with permissions *mode*. Raises FileExistsError if the directory to be created already exists.
os.makedirs(*path*, *mode*)	*ibid*, but will create intermediate directories in *path*.
os.listdir([*path*])	Get all files and directories at *path*. If no *path* is provided, the contents of CWD are returned.
os.remove(*path*)	Remove *path*. CANNOT be used to remove directories.
os.rmdir(*path*)	Remove or delete an *empty* directory. Raises OSError if *path* is not an empty directory. To delete an entire tree (everything on the path), use shutil.rmtree(*path*) (**very, very dangerous!**).
subprocess.Popen(*command*, *shell=False*)[4]	Spawn a child process using *command*. Run as a direct shell command with *shell=True* (**can be dangerous!**). Usage of this command is complex, see `https://docs.python.org/3/library/subprocess.html#security-considerations`
os.rename(*old_path*, *new_path*)	Rename *old_path* to *new_path*. Raises FileNotFoundError if *old_path* cannot be found.
os.path.exists(*path*)	See if *path* exists. Returns boolean.
os.path.getsize(*path*)	Get the size of *path*.

Table 1: Common os module commands.