ngworx.ag

Python! When a shell script is just not good enough.

March 2019
written by Lukas Resch

Every one of us faces repetitive tasks that we usually do not enjoy. Luckily for those of us working in IT most of these tasks can be automated quite easily. For decades the tool for simple automation tasks was the shell. Shell scripts allow you to create automation fast by utilizing available system tools and pipes. Although when it comes to more complex tasks the disadvantages of shell scripts become obvious.

The syntax makes large scripts hard to read, missing language features make complex implementations a mess, error handling is at a level of hopeful at its best, and allowing runtime configuration with command line options often exceeds the actual code in many cases. Furthermore, basic operations that are easy to handle in other languages, such as mathematical, string or date operations are hard to handle in a shell script. Even if you can deal with all these shortcomings when you are asked about portability or whether you could add a graphical user interface to your script.

Shell scripts are Turing complete, so we could just add more shell code to circumvent the shortcomings of shell scripts. But then again sed is also Turing complete and you would probably not argue to write your automation with sed, would you? Sometimes it is just necessary to step back and re-evaluate a tool. Maybe you've reached the end of the scale and it is necessary to replace your tool.

One tool to replace your shell scripts is Python. Python provides an easy to learn but still very powerful programming language. Python uses white space indentation to delimit blocks. While that is criticizable, it also ensures that the resulting code remains easy to read. Python also provides an extensive standard library which already implements a lot of functionality in a portable way. Given you follow common best practices while writing your code, Python also allows you to add a graphical user interface with a small implementation overhead. Even though a line in a shell script turns into a few lines in Python, the improved error handling and the gained type safety make up for that.

Functions

Within Python, it is easy to reuse code in multiple places. In shell scripts, it is possible to create functions and thereby call the same code from multiple locations. However, it is not possible to define the parameters that can be passed to the function. The necessary parameters can therefore only be determined by reading through the function body looking for variables that indicate by their name that they should be passed to the functions. Within Python, functions are defined through their parameters. For example, if we wanted to build a Python function to replace the call to grep in our Python program we could give it the signature as shown in the code below. Thereby anyone calling the function would see that there are two parameters required to call the function. Using proper naming for the parameters also allows the user to understand what the parameters should contain.

```
def grep(pattern, path):
```

It is further possible to provide default values for function parameters. Thereby we could allow our grep implementation to run recursively on the given path if explicitly requested by the user.

```
def grep(pattern, path, recursive=False):
```

Depending on our use case we now could return a list of lines matching the pattern, a dictionary mapping line numbers to line contents, simply print the lines or a combination of the possibilities. Unlike shell scripts, we are not limited to the standard output when passing output from one function to another.

Portability

A problem you might have run into when working with file paths on different platforms is that Unix and Windows use a different path separator. The Python module *os* will give you the separator of the platform your program runs on.

```
import os
print(os.sep)
```

Furthermore, it can also take care of paths completely for you. Thereby you can focus on your program logic and let Python take care of portability.

```
path = os.path.join(os.getcwd(), filename)
directory, file = os.path.split(path)
```

Sometimes it might not be feasible to implement available functionality from an available binary

again in Python. For these situations, it is possible to just call the binary from Python, but handle its output within your Python program. The *subprocess.check_output* function allows you to call any external executable and will return the program's entire output. Be aware that extensive use of the function might reduce the portability of your program.

Error handling

For every use case, we have covered so far there are a few error cases. Within a shell script, you would most likely check the return value of the called binary or run your script with the -e flag set. In the first case, you are prone to lose all your error handling by small changes to the code at a later point in time. For example, adding progress output could sweep away your error checking.

```
mv $FILE /non/existing/path
echo 'Progress 25%'
if [[ $? -ne 0 ]]
then
    echo 'ERROR: cannot move file'
    exit 1
fi
```

Errors are signalled in Python with exceptions and there is no option to miss them. If your program does not catch an exception, the program is terminated and a stack trace shows were the exception came from. In most cases though, it is more desirable to handle the error within the program. Therefore, it is necessary to catch the exception. Depending on the situation you want to explicitly declare which exception you expect. It is also possible to declare multiple exceptions or have a *final* block, which is executed independently of the result.

```
try:
    do_something()
 except:
    rollback()
 finally:
    notify_user()
```

Handling arguments

Another recurring problem is the usage of command line parameters. Some of them may be necessary, while others are optional. Some of them can be provided only once, while others can be given multiple times. And of course, we want to provide up-to-date usage instructions from our program. Within a shell script, we would now need to write a huge amount of code to take care of

that. In Python, we will just use the argparse module to define the parameters. Argparse will not only handle the user input for us and provide us with the result in Python variables with proper types, but also generate usage instruction for us. The user interface for our grep implementation is thereby implemented in just four lines of code.

```
import argparse
parser = argparse.ArgumentParser(description='Simple grep imlementation')
parser.add_argument('-r', action='store_true', help='Recursively grep
through files')
parser.add_argument('pattern', metavar='PATTERN', help='The search
pattern')
parser.add_argument('file', metavar='FILE', help='File to search')
```

Now that our grep has a rather user-friendly interface you might want to share it with your friends. Python provides its own package manager (pip) and includes support building and installing with distutils. Especially with larger projects that carry dependencies, you will want to use setuptools though.

Conclusion

Within this article, I gave you a short introduction to Python and its standard library. Hopefully, it was enough to motivate you to take a deeper look and you will reconsider using shell scripts for your daily automation tasks.

Beside simple scripts, Python can be used for a lot of other things. From building a small CLI client for your REST API to machine learning there is a Python module for every problem. If you want to fly just *import antigravity*!

LikeColman Finnin likes this