The Python language is a trusted constant in the world of DevOps Engineering due to its ability to facilitate automation. Python’s philosophy of “batteries included” means there is a module already built for (nearly) anything you need to accomplish, making constructing automation and moving from development to production a less burdensome process.

Here are several, of the many, amazing Python modules for DevOps, along with some sample code exhibiting some of their basic functionality:

requests

Requests is one of the most popular Python modules (over 30M downloads / week) and is used to make HTTP requests extremely easy. Instead of coding GET, POST, PUT, etc. data in your Python, just use Requests!

Install it

pip install requests

(Throughout this article, I’ll be using pip, although you may need to use pip3, depending on your system).

For this request, we’ll be using the Dog API (https://dog.ceo/dog-api/)

import requests

r = requests.get("https://dog.ceo/api/breeds/image/random")

print(r.json())

{'message': 'https://images.dog.ceo/breeds/hound-afghan/n02088094\_305.jpg', 'status': 'success'}



Here are some of the many supported features of Requests:

* Keep-Alive & Connection Pooling
* Sessions with Cookie Persistence
* Browser-style TLS/SSL Verification
* Basic & Digest Authentication
* Chunked HTTP Requests

Learn More

<https://pypi.org/project/requests/>

os

The os (Operating System) module provides a portable way for us to interact with our…you guessed it…operating system!

There is no need to install OS, as it is included in the standard Python3 library. You just need to import it into your code.

import os

Here are some of the OS modules I’ve found to be useful:

os.**getenv**(*key*, *default=None*)

Returns the value of the environment variable *key* as a string if it exists, or *default* if it doesn’t.

os.getenv, without parentheses, will return *all* environmental variables, which we we can loop through, using the ‘key’ of the key-value pair:

import os

for key in os.environ:

print(f"The key is {key} and the value is {os.environ[key]}")

The key is COLORTERM and the value is truecolor

The key is PYENV\_SHELL and the value is bash

The key is PYENV\_VERSION and the value is 3.9.3

The key is my\_code and the value is abc123

os.**uname**()

Returns information about the current operating system. The return value is an object having five attributes:

* sysname - operating system name
* nodename - name of machine on network (implementation-defined)
* release - operating system release
* version - operating system version
* machine - hardware identifier

import os

print(os.uname())

posix.uname\_result(sysname='Darwin', nodename='Python-iMac.local', release='1.7.5', version='Darwin Kernel Version 19.0.0: Sun Oct 9 20:14:54 PDT 2022; alexstev:xnu-2782.1.97~2/RELEASE\_X86\_64', machine='x86\_64')

os.**getlogin**()

Return the name of the user logged in on the controlling terminal of the process.

import os

print(os.getlogin())

alexstev

Learn More

<https://docs.python.org/3/library/os.html>

sys

The sys module provides functions and variables that are used to manipulate different parts of your Python runtime environment. It allows for operating on the interpreter, as it provides access to the variables and functions that interact strongly with the interpreter.

Here are some of the useful sys modules I’ve used:

sys.**argv**

The list of command line arguments passed to a Python script. argv[0] is the script name

import sys

print("This script name:", sys.argv[0])

print("The arguments passed are:", str(sys.argv))

$python3 modules.py 1 2 3 4

This script name: modules.py

The arguments passed are: ['modules.py', '1', '2', '3', '4']

sys.**version**

A string containing the version number of the Python interpreter and additional information on the compiler used and build number.

sys.version

import sys

print(sys.version)

3.9.3 (main, Nov 25 2022, 01:25:06) [Clang 14.0.0 (clang-1300.0.25.201)]

sys.exc\_info

Useful for exception and error handling, it returns type of the exception, the exception itself, and a [traceback object](https://docs.python.org/3/reference/datamodel.html#traceback-objects) which typically encapsulates the call stack at the point where the exception last occurred.

import sys

bad\_integer = "a"

try:

print(int(bad\_integer))

except Exception:

extype, exc, tb = sys.exc\_info()

print(f"Exception type: {extype}\nException:{exc}\nTraceback:{tb}")

Learn More

<https://docs.python.org/3/library/sys.html>

re

The re (Regular expression) module provides regular expression matching operations where the patterns and strings to be searched can be Unicode strings ([str](https://docs.python.org/3/library/stdtypes.html#str)) as well as 8-bit strings ([bytes](https://docs.python.org/3/library/stdtypes.html#bytes)).

re.sub()

The syntax is re.sub(pattern, replace, string)

This method returns a string where matched occurrences are replaced with the content of the ‘replace’ variable.

import re

old\_url = "www.website.com"

new\_url = re.sub("www.", "https://", old\_url)

print(new\_url)

**re.findall()**

This method returns a list of strings containing all matches.

import re

my\_string = "We have an inventory of 10 planes and 20 office buildings."

digits\_pattern = "\d+"

result = re.findall(digits\_pattern, my\_string)

print(result)

['10', '20']

**re.sub()**

The syntax of re.sub() is: re.sub(pattern, replace, string)

This method returns a string where matched occurrences are replaced with the content of replace variable.

import re

# multiline string

device\_config = "Server:283474 Node:238474 ID:3833873"

# matches all whitespace characters

pattern = "\s+"

# empty string

replace = ", "

pretty\_device\_config = re.sub(pattern, replace, device\_config)

print(pretty\_device\_config)

Server:283474, Node:238474, ID:3833873

Learn More

<https://docs.python.org/3/library/re.html?highlight=re#module-re>

Fabric

Fabric is a high level Python library designed to execute shell commands remotely over SSH, yielding useful Python objects in return. Fabric builds on top of [Invoke](https://pyinvoke.org/) (subprocess command execution & command-line features) and [Paramiko](https://paramiko.org/) (SSH protocol implementation), extending their APIs to complement one another, while providing additional functionality.

The most common ways of installing Fabric is via, pip, easy\_install or via

the operating system's package manager:

pip install fabric2

sudo easy\_install fabric2

sudo apt-get install fabric2

Fabric provides the fab command which reads its configuration from a file, fabfile.py. The file should be in the directory from which the command is run. A standard fabfile contains the functions to be executed on a remote host or a group of remote hosts.

Let’s see how we can us fabric for connecting to a remote host and executing commands in it.

from fabric import Connection, task

from fabric import Connection, task

@task

def deploy(ctx):

with Connection(

"sandbox-iosxe-latest-1.cisco.com",

user="developer",

connect\_kwargs={"password": "C1sco12345"},

) as c:

c.run("sh ip int br")

fab deploy

Interface IP-Address OK? Method Status Protocol

GigabitEthernet1 10.10.20.48 YES NVRAM up up

GigabitEthernet1.10 10.0.0.10 YES other up up

GigabitEthernet1.20 20.0.0.20 YES manual up up

(response truncated for brevity)

We can also any number of Git commands, such as Pull or Commit, or Push, as well as the standard Docker commands. Image how much you can simplify your CI/CD pipeline with these!

Learn More

<https://www.fabfile.org/>

Subprocess

Platform

BeautifulSoup4

Smtplib