

#90DaysOfDevOps Challenge - Day 15 - Python Libraries for DevOps

Welcome to Day 15 of the #90DaysOfDevOps challenge. As a DevOps Engineer, one of the key skills you should possess is the ability to effectively parse and manipulate files in various formats. Whether it's working with text files, parsing **JSON** data, or handling **YAML** configuration files, being proficient in file parsing is essential for successful DevOps workflows.

In Python, several powerful libraries can greatly assist you in these tasks, ensuring efficient and streamlined file handling. Let's explore some of these essential libraries and understand their significance in the realm of DevOps.

Essential Python Libraries for DevOps: Handling File Parsing and Manipulation

1. OS Library: Operating System Interactions

The **os** library in Python provides a range of functions for interacting with the underlying operating system. As a DevOps Engineer, you'll frequently need to perform operations like file and directory manipulation, environment variable management, and executing system commands. The **os** library simplifies these tasks by offering functions such as **os.path** for path manipulation, **os.listdir** for listing directory contents, and **os.environ** for accessing and modifying environment variables.

```
import os

# Example of path manipulation
current_directory = os.getcwd()
print("Current Directory:", current_directory)

# Example of listing directory contents

print("Files in Directory:", files)

# Example of accessing environment variables
home_directory = os.environ.get("HOME")
print("Home Directory:", home_directory)
```

The code imports the **os** library for operating system interactions. It demonstrates path manipulation by retrieving the current directory using **os.getcwd()** and listing the files in the directory using **os.listdir()**. It also showcases accessing environment variables by retrieving the value of the **"HOME"** variable using **os.environ.get()**. The results are printed to the console.

2. SYS Library: System-Specific Parameters and Functions

The **sys** library allows you to access system-specific parameters and functions in Python. It provides useful functionalities such as command-line argument handling (**sys.argv**), standard I/O redirection (**sys.stdin**

and `sys.stdout`), and system-specific configuration details (`sys.platform`). These features are invaluable for building robust and versatile DevOps scripts and utilities.

```
import sys

# Example of command-line argument handling
script_name = sys.argv[0]
arguments = sys.argv[1:]
print("Script Name:", script_name)
print("Arguments:", arguments)

# Example of standard I/O redirection
sys.stdout.write("Hello, DevOps Engineers!\n")
sys.stderr.write("Error: Something went wrong!\n")

# Example of system-specific configuration details
platform = sys.platform
print("Platform:", platform)
```

The code imports the `sys` library for accessing system-specific parameters and functions. It demonstrates command-line argument handling by assigning the script name to `sys.argv[0]` and the arguments to `sys.argv[1:]`. It showcases standard I/O redirection by writing a message to `sys.stdout` and an error message to `sys.stderr`. It also retrieves the system's platform using `sys.platform` and prints it to the console.

3. JSON Library: Working with JSON Data

JSON (JavaScript Object Notation) is a widely used data interchange format in modern applications. As a DevOps Engineer, you'll often encounter JSON files or interact with APIs that return JSON responses. The `json` library in Python provides functions for encoding Python objects into JSON and decoding JSON data into Python objects. It simplifies the process of reading, writing, and manipulating JSON data, enabling seamless integration with other DevOps tasks and systems.

```
import json

# Example of encoding Python objects into JSON
data = {
    "name": "John Doe",
    "age": 30,
    "city": "New York"
}
json_data = json.dumps(data)
print("JSON Data:", json_data)
```

The code imports the `json` library and demonstrates how to encode a Python dictionary into a JSON string. It creates a dictionary `data` with name, age, and city information, then uses `json.dumps()` to convert it into a JSON format. Finally, it prints the JSON data.

4. YAML Library: Handling YAML Configuration Files

YAML (YAML Ain't Markup Language) is a human-readable data serialization format commonly used for configuration files. DevOps Engineers frequently work with YAML files to configure infrastructure, define deployment pipelines, and store application settings. The `yaml` library in Python allows you to easily load YAML files, convert them into Python data structures, and vice versa. This library simplifies the task of reading and writing YAML files, making it effortless to work with YAML-based configurations in your DevOps workflows.

```
import yaml

# Example YAML data
yaml_data = """
name: John Doe
age: 30
city: New York
"""

# Load YAML data
data = yaml.load(yaml_data, Loader=yaml.SafeLoader)

# Accessing values from the YAML data
name = data["name"]
age = data["age"]
city = data["city"]

# Print the values
print("Name:", name)
print("Age:", age)
print("City:", city)
```

The code imports the `yaml` library and demonstrates how to load YAML data and access its values. It defines a YAML string `yaml_data` with name, age, and city information. The code then loads the YAML data using `yaml.load()` and retrieves the values using dictionary access. Finally, it prints the name, age, and city values.

These are just a few examples of the numerous libraries available in Python that are indispensable for DevOps Engineers. Depending on your specific requirements and project needs, you may also find other libraries such as `requests` for HTTP requests, `paramiko` for SSH connectivity, and `docker` for interacting with Docker containers is extremely useful.

Task 1 - Create a Dictionary in Python and write it to a JSON File.

```
import json

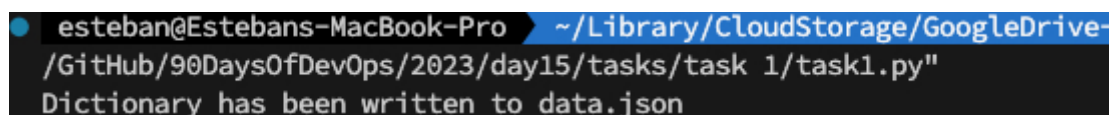
# Define a dictionary
dictionary = {
    "name": "John Doe",
```

```
    "age": 30,
    "city": "New York"
}

# Open a file in write mode and use json.dump() to write the dictionary to
the file
with open("data.json", "w") as file:
    json.dump(dictionary, file, indent=4)
    # The indent parameter is used to format the JSON data with
    indentation for better readability

print("Dictionary has been written to data.json")
```

In this code, the `json` library is imported. Then, a `dictionary` named `dictionary` is defined, containing some key-value pairs. The `open()` function is used to open a file named `"data.json"` in write mode with the `"w"` mode specifier. The `json.dump()` function is called to write the `dictionary` to the file. The `indent=4` parameter is used to format the JSON data with an indentation of 4 spaces per level for better readability. Finally, a message is printed to confirm that the dictionary has been successfully written to the `"data.json"` file.



```
esteban@Estebans-MacBook-Pro ~/Library/CloudStorage/GoogleDrive-
/GitHub/90DaysOfDevOps/2023/day15/tasks/task 1/task1.py
Dictionary has been written to data.json
```

Task 2 - Read a JSON file `services.json` kept in this folder and print the service names of every cloud service provider.

```
import json

# Open the 'services.json' file
with open('services.json') as file:
    # Load the JSON data into a Python dictionary
    data = json.load(file)

# Access the 'name' attribute of the 'aws' service
print("aws: ", data['services']['aws']['name'])

# Access the 'name' attribute of the 'azure' service
print("azure: ", data['services']['azure']['name'])

# Access the 'name' attribute of the 'gcp' service
print("gcp: ", data['services']['gcp']['name'])
```

The code reads the `'services.json'` file and loads its content into the `data` dictionary. It then accesses the `'name'` attribute of the `'aws'`, `'azure'`, and `'gcp'` services within the `'services'` dictionary and prints them along with the corresponding service names.

```
esteban@Estebans-MacBook-Pro ~/Library/CloudStorage/GoogleDrive-moreno.com/My Drive/GitHub/90DaysOfDevOps/2023/day15/tasks/task 2/task2.py"
aws: EC2
azure: VM
gcp: Compute Engine
```

Task 3 - Read YAML file using Python, file `services.yaml` and read the contents to convert YAML to JSON

```
import yaml
import json

# Read the YAML file
with open('services.yaml') as file:
    yaml_data = yaml.safe_load(file)

# Convert YAML to JSON
json_data = json.dumps(yaml_data)

# Write the JSON data to a file
with open('task3.json', 'w') as file:
    file.write(json_data)

print("Converted YAML to JSON and saved to task3.json")
```

```
esteban@Estebans-MacBook-Pro ~/Library/CloudStorage/GoogleDrive-morenoramir.com/My Drive/GitHub/90DaysOfDevOps/2023/day15/tasks/task 3/task3.py"
Converted YAML to JSON and saved to task3.json
```

Congratulations on completing Day 15 of the #90DaysOfDevOps challenge. Today, we explored essential Python libraries for DevOps, focusing on handling file parsing and manipulation. We learned about libraries like `os`, `sys`, `json`, and `yaml` that are commonly used by DevOps Engineers in their day-to-day tasks.

Stay tuned for Day 16 of the #90DaysOfDevOps challenge, where we will dive into Docker for DevOps Engineers. Docker is a powerful tool for containerization, allowing you to build, deploy, and manage applications in a consistent and scalable manner. We will explore Docker concepts, commands, and best practices that will enhance your DevOps skills.