PW SKILLS | DevOps and Cloud Computing

Python for DevOps: Object-Oriented Programming and Libraries

Objective

- Understand the principles of object-oriented programming (OOP) in Python.
- Learn how to use classes, objects, and inheritance in Python programs.
- Explore Python libraries that simplify DevOps tasks.
- Use regular expressions to perform pattern matching and text processing.

PW SKILLS

Introducing OOP Concepts

Object-Oriented Programming (OOP)

- Object-oriented programming (OOP) is a programming paradigm based on "objects" that contain data in the form of fields (attributes) and code in the form of procedures (methods).
- OOP focuses on creating reusable code by grouping data and methods that operate on that data into objects.
- The four main principles of OOP are encapsulation, abstraction, inheritance, and polymorphism.

Object-Oriented Programming (OOP)

Classes and Objects

- **Class A** class is a blueprint or a template for creating objects. It defines the attributes (data) and methods (behavior) that the objects of that class will have.
- **Object** An object is an instance of a class. It is a concrete entity that exists in memory and has its own unique state (values of attributes).

For example, consider a class called "Server". It might have attributes like ‘server ID’, ‘Ip address', ‘status’, and ‘memory usage’ and methods like ‘start()’, ‘stop()’, ‘restart()’, and ‘update status()’.

```
class Server:
    def __init__(self, serverID, IP_address):
        self.serverID = serverID
        self.IP_address = IP_address
        self.status = "offline"
        self.memory_usage = 0

    def start(self):
        self.status = "online"
        print(f"Server {self.serverID} started")

    def stop(self):
        self.status = "offline"
        print(f"Server {self.serverID} stopped")

    def restart(self):
        self.stop()
        self.start()
        print(f"Server {self.serverID} restarted")

    def update_status(self, new_status):
        self.status = new_status
        print(f"Server {self.serverID} status updated to {self.status}")

# Creating objects (instances) of the Server class
server1 = Server("SRV001", "192.168.1.100")
server2 = Server("SRV002", "192.168.1.101")

# Accessing attributes and calling methods
print(server1.status)  # Output: offline
server1.start()         # Output: Server SRV001 started
print(server1.status)  # Output: online
server2.update_status("busy") # Output: Server SRV002 status updated to busy
```

Object-Oriented Programming (OOP)

In the above example, ‘Server’ is a class, while ‘server1’ and ‘server2’ are objects (instances) of the ‘Server’ class. Each server object has its own ‘server ID’, ‘IP address’, ‘status’, and ‘memory usage’.

OOP Principles:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

PW SKILLS

Demonstrating creating a class, defining attributes and methods, and creating objects

Let’s do it

Okay, here's a short demonstration of creating a class, defining attributes and methods, and creating objects using a simplified ‘Dog’ example:

```python
# Define a class called Dog
class Dog:
    # Constructor - initializes attributes when a Dog object is created
    def __init__(self, name, breed):
        self.name = name  # Attribute: dog's name
        self.breed = breed # Attribute: dog's breed
        self.is_sleeping = False  # Attribute:  Default state

    # Method:  Make the dog bark
    def bark(self):
        return "Woof!"

    # Method: Put the dog to sleep
    def sleep(self):
      self.is_sleeping = True
      return f"{self.name} is now sleeping."

    # Method: Wake up the dog
    def wake_up(self):
      self.is_sleeping = False
      return f"{self.name} is now awake!"

# Create objects (instances) of the Dog class
my_dog = Dog("Buddy", "Golden Retriever")
your_dog = Dog("Lucy", "Poodle")

# Access attributes and call methods
print(f"{my_dog.name} is a {my_dog.breed}")  # Output: Buddy is a Golden Retriever
print(your_dog.bark())  # Output: Woof!
print(my_dog.sleep())  # Output: Buddy is now sleeping.
print(my_dog.wake_up()) # Output: Buddy is now awake!
```

PW SKILLS

Explaining Inheritance

Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows a class (known as the child or subclass) to inherit properties and methods from another class (known as the parent or superclass).

Example of Inheritance

Parent Class: Animal

- The ‘Animal’ class serves as the base class, defining common attributes And methods for all animals.

```python
class Animal:
    def __init__(self, name, species):
        self.name = name        # Attribute: Name of the animal
        self.species = species  # Attribute: Species of the animal

    def speak(self):
        return "Some generic animal sound"

    def info(self):
        return f"{self.name} is a {self.species}."
```

Inheritance

Child Class: Dog

- The ‘Dog’ class inherits from the ‘Animal’ class and provides specific implementations.

```
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, "Dog")  # Call the parent
class's constructor
        self.breed = breed              # Additional attribute for Dog

    def speak(self):  # Method overriding
        return "Woof!"

    def info(self):  # Method overriding to include breed information
        return f"{self.name} is a {self.breed} dog."
```

Inheritance

Creating Instances and Demonstrating Inheritance

Now, let's create instances of both classes and demonstrate how inheritance works.

```
# Create an instance of the Animal class
generic_animal = Animal("Generic Animal", "Unknown")
print(generic_animal.info())  # Output: Generic Animal is a Unknown.
print(generic_animal.speak())  # Output: Some generic animal sound

# Create an instance of the Dog class
dog = Dog("Buddy", "Golden Retriever")
print(dog.info())             # Output: Buddy is a Golden Retriever dog.
print(dog.speak())            # Output: Woof!
```

Inheritance

Method Overriding:

- When calling ‘speak()’ on an instance of ‘Dog’, it executes the overridden method in the ‘Dog’ class instead of the one in the ‘Animal’ class.
- Similarly, calling ‘info()’ on a ‘Dog’ instance provides more specific information than calling it on a generic animal.

PW SKILLS

Providing an overview of popular DevOps-related Python libraries

Overview

Here's an overview of popular Python libraries used in DevOps:

- OS module: This module allows interaction with the operating system, enabling tasks such as file and directory manipulation.
- subprocess: Along with the os module, subprocess is used for executing shell commands and scripts.
- boto3: This is the AWS SDK for Python, which allows you to manage AWS resources. It enables DevOps engineers to automate the management of AWS services, such as EC2, S3, and Lambda, directly from Python scripts.
- Paramiko: Paramiko is an SSH library that facilitates secure communication with remote servers. It enables you to execute commands, transfer files, and manage remote servers securely via SSH, making it crucial for automating tasks across a network of machines.
- Ansible: Ansible is a powerful automation tool that is particularly useful for system management duties. It is coded purely in Python.

PW SKILLS

Demonstrating a simple use case for one of these libraries

Let's do it

Checking Disk Usage with Python

You can utilize the 'shutil.disk usage()' function to retrieve information about the total, used, and free disk space for a specified path. Here's a brief example:

```python
import os
import shutil

def check_disk_usage(disk):
    # Get disk usage statistics
    total, used, free = shutil.disk_usage(disk)
    return total, used, free

# Example usage: check disk usage for the root directory
disk_stats = check_disk_usage('/')
print(disk_stats)
```

Let’s do it

Execution Results:

When you run this code, it will provide output similar to:

```
(9.087889e+09, 3.782631e+09, 5.288481e+09)
```

This output represents:

- Total disk space: approximately 9.09 GB
- Used disk space: approximately 3.78 GB
- Free disk space: approximately 5.29 GB

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

PW SKILLS

Introducing regular expressions and their role

Regular Expressions

- Regular expressions (regex) are special text strings that serve as patterns for describing and matching specific amounts of text.
- They provide a concise and flexible means to manipulate text data.
- Regular expressions are a powerful tool that can save time and effort when extracting information, validating user input, or parsing text files.

Regular Expressions

Key aspects of regular expressions:

- Pattern Matching: Regexes are used to find or match patterns present in text. They define a search pattern, allowing you to easily match and extract specific substrings from text data.
- Text Processing: They are used in text processing for defining patterns that can match specific sequences of characters within a target string.
- Flexibility: Regular expressions allow expressing complex patterns in a compact and readable form.
- Versatility: They can be used for find-and-replace operations, data transformation tasks, and validating input data like emails and phone numbers.
- Efficiency: When used correctly, regexes can significantly speed up text processing tasks

PW SKILLS

Explaining basic components of regex

Basic components of regex

Regular expressions consist of several basic components:

Patterns:

- Literals: Ordinary characters that match themselves (e.g., 'a' matches 'a').
- Metacharacters: Special characters with specific meanings (e.g., ‘.’ matches any character, ‘(backslash)d’ matches any digit).
- Quantifiers: Specify the number of occurrences (e.g., ‘*’ for zero or more, ‘+’ for one or more, ‘{n}’ for exactly n occurrences).

Basic components of regex

Anchors:

- ‘^’: Matches the start of a string.
- ‘$’: Matches the end of a string.

Groups and Character Classes:

- Groups: Enclosed in parentheses ‘()’, they capture parts of the match (e.g., ‘(abc)’ captures 'abc').
- Character Classes: Defined by square brackets ‘[ ]’, they match any one of the enclosed characters (e.g., ‘[a-zA-Z]’ matches any letter).

PW SKILLS

Demonstrating using Python's re module for common tasks

Let’s do it

Here's a demonstration of common tasks using Python's ‘re’ module for regular expressions:

1. Matching Strings

Using ‘re.match’ and ‘re.search’:

```
import re

# Example string
text = 'Hello, welcome to the world of regex!'

# Using re.match to check if the string starts with 'Hello'
match_result = re.match(r'Hello', text)

# Using re.search to find 'world' in the string
search_result = re.search(r'world', text)

match_result, search_result
```

Let’s do it

Execution Results:

- match result: A match object indicating that the string starts with 'Hello'.
- search result: A match object indicating that 'world' is found at position 22.

2. Finding Patterns

Using ‘re.findall’:

```
import re

# Example string
text = 'The rain in Spain stays mainly in the plain.'

# Using re.findall to find all occurrences of 'ain'
findall_result = re.findall(r'ain', text)

findall_result
```

Let’s do it

Execution Results:

- ‘findall result’: A list containing all occurrences of 'ain': ['ain', 'ain', 'ain', 'ain'].

3. Replacing Text

Using ‘re.sub

```
import re

# Example string
text = 'I love apples. Apples are my favorite fruit.'

# Using re.sub to replace 'apples' with 'oranges'
sub_result = re.sub(r'apples', 'oranges', text, flags=re.IGNORECASE)

sub_result
```

Execution Results:

- sub result: The modified string: "I love oranges. oranges are my favorite fruit."

PW SKILLS

Time for Case Study

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this session
- Review your lecture notes
- Note down questions and queries regarding this session and consult the teaching assistants

Thanks !