Python For DevOps

Enhancing Automation and Efficiency

**Index**

1. Overview of Python in DevOps
2. Data Types
3. Keywords and Variables
4. Functions, Modules and Packages
5. CLI Arguments & Env Variables
6. Operators in Python
7. Condition Handlings
8. List & Tuples
9. Loops
10. Working with Lists
11. Dictionaries & Sets
12. File Operations
13. Boto3 Overview
14. JIRA & GitHub Integration
15. Basic Interview Questions

# Overview of DevOps

# Overview of Python in DevOps

Briefly describe the DevOps culture and its goals—emphasizing automation, collaboration between development and operations teams, and continuous integration/delivery.

# Python in DevOps

Highlight why Python has become a go-to scripting language for DevOps engineers. Its simplicity, flexibility, and vast ecosystem of libraries make it ideal for DevOps automation and integrations.

# Four Pillers of Any Programming Language:

* Keywords
* Data Types
* Operators
* Logic Reasoning Skills

# Key Advantages of Python in DevOps

* Ease of Use:
  + Python’s readable syntax and ease of learning make it accessible for both

beginner and experienced DevOps engineers

* Cross-Platform Compatibility:
  + Python scripts run on multiple operating systems, including Linux, Windows, and macOS, ensuring consistency across environments.
* Rich Library Ecosystem:
  + Python has extensive libraries for interacting with APIs, automating infrastructure, and working with data, making it suitable for various DevOps use cases.
* Community and Support:
  + Python’s strong community support ensures fast problem-solving and access to many resources and tools.

# Core Use Cases of Python in DevOps

1. Automation of CI/CD Pipelines

Automating deployment, testing, and version control using tools like Jenkins, GitLab CI, or CircleCI with Python scripts.

1. Infrastructure as Code (IaC)

Provisioning and managing cloud resources and infrastructure using Python with tools like Terraform and AWS Boto3 SDK.

1. Configuration Management

Managing configurations across different environments using Python to ensure uniformity in deployment pipelines.

1. System Monitoring & Alerts

Creating monitoring scripts for logging, alerting, and integrating with platforms like Nagios, Prometheus, or Grafana.

1. Log Management and Analysis

Parsing and analyzing logs with Python for real-time monitoring and error tracking.

1. API Integration

Utilizing Python for interacting with external services via REST APIs, like integrating with cloud providers (AWS, Azure) or monitoring services (DataDog, Splunk).

1. Cloud Automation

Using Python scripts to automate tasks in AWS, Google Cloud, and Azure like resource provisioning, scaling, and managing deployments.

1. Container Management and Orchestration:

Automating the management of Docker containers and Kubernetes clusters through Python scripts and API integrations.

# Tools and Libraries for Python in DevOps

* Boto3: Automates AWS services for tasks like creating EC2 instances, S3 buckets, or managing RDS databases.
* Ansible: Python-based tool that automates configuration management, software deployment, and orchestration.
* Requests: Simplifies interacting with APIs, useful for DevOps tasks like monitoring or cloud service management.
* Kubernetes Python Client: Automates the management of Kubernetes clusters through API interactions.

# Best Practices for Using Python in DevOps

* + Modularity:
    - Write modular Python code that can be reused across multiple scripts or projects.
  + Error Handling:
    - Implement proper error handling to ensure scripts can gracefully recover from failures or report meaningful issues.
  + Version Control:
    - Use Git to manage and version Python scripts, ensuring transparency and tracking of changes in automation scripts.
  + Logging and Monitoring:
    - Implement logging within Python scripts to keep track of script performance and errors during execution.
  + Security Considerations:
    - Ensure Python scripts handling sensitive data (e.g., credentials or tokens) follow security best practices, like encryption and secure storage.

# Sample Python Script:

File Save format with .py extension. test.py

print("Hello, World!")

python test.py  Output: Hello, World!

# What is Data Type:

**Data Types**

* + Python provides several built-in data types to store and manipulate data.
  + **Python is a - Dynamically Type Programming language** where we are not explicitly mentioning the data types definition.

Below are the data types in the Python:

1. Numeric Types
2. Sequence Types
3. Mapping Type
4. Set Types
5. Boolean Type
6. Binary Types
7. None Type
8. **Numeric Types:**
   1. int: Represents integers, which are whole numbers, positive or negative.
      1. a = 10 # Example of an integer
   2. float: Represents floating-point numbers (decimals).
      1. b = 10.5 # Example of a float
   3. complex: Used for complex numbers, consisting of a real and an imaginary part.
      1. c = 2 + 3j # Example of a complex number
9. **Sequence Types**
   1. str: Represents strings, which are sequences of Unicode characters.
      1. name = "DevOps Automation" # Example of a string
   2. list: A mutable, ordered collection of items, which can be of any data type.
      1. fruits = ["apple", "banana", "cherry"] # Example of a list
   3. tuple: An immutable, ordered collection of items.
      1. coordinates = (10, 20) # Example of a tuple
10. **Dictionary Type**
    1. dict: A collection of key-value pairs, where the keys must be unique.
       1. person = {"name": "John", "age": 30} # Example of a dictionary
11. **Set Types**
    1. set: An unordered collection of unique elements.
       1. numbers = {1, 2, 3, 4} # Example of a set
    2. frozenset: An immutable version of a set.
       1. frozen\_numbers = frozenset({1, 2, 3, 4}) # Example of a frozenset
12. **Boolean Type**
    1. bool: Represents one of two values: `True` or `False`.
       1. is\_valid = True # Example of a boolean
13. **Binary Types**
    1. bytes: Immutable sequence of bytes.
       1. byte\_data = b"hello" # Example of bytes
    2. bytearray: A mutable sequence of bytes.
       1. byte\_array = bytearray(5) # Example of a bytearray with 5 empty byte
14. **None Type**
    1. NoneType: Represents the absence of a value.
       1. result = None # Example of None type

**Some Python Code Examples: # Integer variables**

num1 **=** 10

num2 **=** 5

# Integer Division result1 **=** num1 **//** num2

**print(**"Integer Division:"**,** result1**)** # Modulus (Remainder)

result2 **=** num1 **%** num2

**print(**"Modulus (Remainder):"**,** result2**)** # Absolute Value

result3 **= abs(-**7**) print(**"Absolute Value:"**,** result3

**# Float variables**

num1 **=** 5.0

num2 **=** 2.5

# Basic Arithmetic result1 **=** num1 **+** num2

**print(**"Addition:"**,** result1**)**

**#String Examples**

text **=** "Python is awesome" length **= len(**text**)**

**print(**"Length of the string:"**,** length**)**

**Keywords:**

# Keywords and Variables

Keywords are reserved words in Python that have a predefined meaning and cannot be used for anything other than their intended purpose, such as naming variables or functions.

They are part of the language syntax and serve specific roles like defining functions, control flow, or handling exceptions.

For example:

* **Control flow:** if, else, elif, for, while
* **Function definition:** def, return
* **Boolean values:** True, False
* **Exception handling:** try, except, finally, raise

We can see all the list of available keywords using below simple python program.

import keyword print(keyword.kwlist)

Results:

['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is',

'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

**Variables**:

Symbolic names that reference values stored in memory. In Python, variables do not need explicit declaration and are created when you assign a value to them. Variable names can contain letters, numbers, and underscores, but they **cannot start with a number** and **cannot use keywords** as names.

Rules for Variables:

* Must start with a letter or underscore (\_).
* Can contain letters, numbers, and underscores (\_).
* Case-sensitive, so myVar and myvar are different variables.
* Cannot use Python keywords as variable names.

Example:

# Creating variables

name = "Alice" # String variable

age = 30 # Integer variable is\_active = True # Boolean variable height = 5.6 # Float variable

**Variables Scope:**

The scope of a variable refers to the region of the program where that variable is accessible. Variables can have different scopes depending on where they are declared. Python has four types of variable scopes.

**LEGB Rule**

Python follows the **LEGB rule** to resolve variable names:

1. **Local:** The interpreter first checks for the variable in the current local scope.
2. **Enclosing:** If not found, it checks the enclosing (nonlocal) scope.
3. **Global:** Then it checks the global scope.
4. **Built-in:** Finally, it checks the built-in scope. Example for the LEGB rule:

x = "global" def outer():

x = "enclosing"

def inner():

x = "local" print(x) # "local"

inner()

print(x) # "enclosing"

outer()

print(x) # "global"

# Local Scope

A variable defined inside a function is called a local variable. It is accessible only within that function and not outside of it.

def my\_function():

x = 10 # Local variable print(x)

my\_function()

# print(x) # Error: x is not defined outside the function

# Enclosing (Nonlocal) Scope

Variables in an enclosing scope are variables from the outer function that can be accessed by inner functions. The nonlocal keyword is used to modify them inside nested functions.

def outer\_function():

x = 10 # Enclosing variable def inner\_function():

nonlocal x # Modify the enclosing variable x = 20

print("Inner:", x) inner\_function() print("Outer:", x)

outer\_function()

# Global Scope

A global variable is defined at the top level of a script or module and is accessible from any part of the program. To modify a global variable inside a function, the global keyword is used.

x = 5 # Global variable def my\_function():

global x # Modify the global variable

x = 10

my\_function()

print(x) # Output: 10

# Built-in Scope

The built-in scope contains names that are preloaded in Python, such as print(), len(), etc. These variables are accessible globally unless shadowed by local or global variables.

print(len([1, 2, 3])) # 'len' is a built-in function

**Functions:**

# Functions, Modules and Packages

In Python function is a block of reusable code designed to perform a specific task. Functions allow for more organized, modular, and maintainable code.

Syntax:

def function\_name(parameters): # Block of code

return value

Example:

def greet(name):

return f"Hello, {name}!"

print(greet("Alice"))

Note:

* **Arguments/Parameters**: Data passed into functions.
* **Return Statement**: A function can return values using return.
* Functions can be **called** multiple times with different inputs, promoting code reuse.

**Modules:**

A module is a file containing Python definitions and statements (e.g., functions, classes, or variables) that can be imported and used in other Python scripts.

Creating a Module:

Any Python file (e.g., my\_module.py) is a module. Suppose we have a file named my\_module.py:

# my\_module.py def add(a, b):

return a + b

def subtract(a, b): return a - b

Importing a Module:

You can import and use the module's functions in another script:

# main.py

import my\_module

result = my\_module.add(5, 3) print(result) # Output: 8

Importing Specific Functions:

from my\_module import add print(add(5, 3)) # Output: 8

Python Standard Library:

Python comes with many pre-built modules (e.g., math, os, sys) that you can import and use without writing your own.

import math import os import json

print(math.sqrt(16)) # Output: 4.0

Packages:

A package is a collection of Python modules organized in directories. Each package contains an init .py file, which signifies that the directory is a Python package. Packages help you organize related modules into a hierarchy.

# Creating a Package:

The structure of a package might look like this:

my\_package/

init .py # Initializes the package module1.py # A module inside the package module2.py # Another module

Example of Package:

my\_package/module1.py

def greet():

return "Hello from module1"

my\_package/module2.py

def farewell():

return "Goodbye from module2"

my\_package/init.py This file can be empty, or it can define symbols that will be available when you import the package.

Importing a Package:

# main.py

from my\_package import module1, module2 print(module1.greet()) # Output: Hello from module1 print(module2.farewell()) # Output: Goodbye from module2

# Installing External Packages:

You can install third-party packages using pip, Python's package manager. pip install package\_name

For example, installing the popular requests package: pip install requests

You can then use it in your program:

import requests

response = requests.get("https://api.github.com") print(response.status\_code)

Function Example:

|  |
| --- |
| num1 = 10  num2 = 20  def addition():  add = num1 + num2 print(add)  def substraction(): sub = num2 - num1 print(sub)  def multipication(): mul = num1 \* num2 print(mul)  addition() substraction() multipication() |
| #Another Way Writing the Function:  def addition(num1,num2): add = num1 + num2 return add  def subtract(num1,num2): sub = num2 - num1 return sub  def multipication(num1,num2): mul = num1 \* num2  return mul  print(addition(50,10)) print(subtract(5,10)) print(multipication(10,20)) |

Module Example:

import function\_test as func1 func1.addition()

# CLI Arguments & Env Variables

**Command Line Arguments** in Python allow a user to provide input to a Python script when it is executed from the command line. This is useful for passing options, filenames, or parameters to scripts at runtime.

In Python, command line arguments are stored in the sys.argv list, provided by the sys module.

# Using sys.argv

The sys.argv list contains the arguments passed to the script:

* sys.argv[0] is the name of the script itself.
* sys.argv[1], sys.argv[2], etc., represent the arguments passed to the script. Example:

# script.py import sys

# sys.argv contains the list of command line arguments print("Script Name:", sys.argv[0])

if len(sys.argv) > 1:

print("Arguments passed:", sys.argv[1:]) else:

print("No arguments provided.")

**Running the Script from the Command Line:**

$ python script.py arg1 arg2 arg3

**Output:**

Script Name: script.py

Arguments passed: ['arg1', 'arg2', 'arg3']

**Environment variables** are dynamic variables maintained by the operating system that can affect the way running processes behave. They are often used for configuration settings, like database credentials, API keys, or system paths, and provide a way to pass information into a program without hardcoding it.

* In Python, environment variables can be accessed using the **os** module.
* You can use the os.environ dictionary to access environment variables.

Example:

Defining the Environment variables like export DB\_HOST=”localhost”

in Python program:

import os print(os.getenv("DB\_HOST"))

We can define like below as well:

import os

db\_host = os.getenv('DB\_HOST', 'localhost') db\_port = os.getenv('DB\_PORT', ‘3306’) db\_user = os.getenv('DB\_USER', 'admin')

db\_password = os.getenv('DB\_PASSWORD', 'password') print(f"Connecting to database at {db\_host}:{db\_port} as {db\_user}")

On Linux/MacOS (Bash):

export DATABASE\_URL="postgres://user:pass@localhost:5432/mydb"

On Windows (Command Prompt):

set DATABASE\_URL=postgres://user:pass@localhost:5432/mydb

# Operators in Python

Operators are symbols that perform operations on variables and values. Python supports various types of operators, categorized based on the type of operations they perform.

Here are the main types of operators in Python:

1. Arithmetic: **+, -, \*, /, //, %, \*\***
2. Comparison: **==, !=, >, <, >=, <=**
3. Logical: **and, or, not**
4. Assignment: **=, +=, -=, \*=, /=, //=, %=, \*\*=**
5. Bitwise: **&, |, ^, ~, <<, >>**
6. Identity: **is, is not**
7. Membership: **in, not in**

# Arithmetic Operators

- Arithmetic operators are used for performing basic mathematical operations.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition | 3 + 2 = 5 |
| - | Subtraction | 3 - 2 = 1 |
| \* | Multiplication | 3 \* 2 = 6 |
| / | Division (float result) | 3 / 2 = 1.5 |
| // | Floor Division (integer) | 3 // 2 = 1 |
| % | Modulus (remainder) | 3 % 2 = 1 |
| \*\* | Exponentiation (power) | 3 \*\* 2 = 9 |

Example:

a = 5

b = 3

print(a + b) # 8 print(a // b) # 1 print(a \*\* b) # 125

# Comparison (Relational) Operators

These operators are used to compare values. They return True or False.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Equal to | 3 == 2 → False |
| != | Not equal to | 3 != 2 → True |
| > | Greater than | 3 > 2 → True |
| < | Less than | 3 < 2 → False |
| >= | Greater than or equal to | 3 >= 2 → True |
| <= | Less than or equal to | 3 <= 2 → False |

Example:

a = 5

b = 3

print(a > b) # True print(a == b) # False

# Logical Operators

Logical operators are used to perform logical operations on expressions and return True or False.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and | True if both conditions are True | (3 > 2) and (4 > 3) → True |
| or | True if at least one condition is True | (3 > 2) or (4 < 3) → True |
| not | Inverts the boolean value | not(3 > 2) → False |

Example:

x = True y = False

print(x and y) # False print(x or y) # True print(not x) # False

# Assignment Operators

These operators are used to assign values to variables. They can also perform arithmetic operations during assignment.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Assigns value | a = 5 |
| += | Add and assign | a += 2 (a = a + 2) |
| -= | Subtract and assign | a -= 2 (a = a - 2) |
| \*= | Multiply and assign | a \*= 2 (a = a \* 2) |
| /= | Divide and assign (float) | a /= 2 (a = a / 2) |
| //= | Floor divide and assign | a //= 2 |
| %= | Modulus and assign | a %= 2 |
| \*\*= | Exponentiate and assign | a \*\*= 2 |

Example:

a = 5

a += 3 # a = a + 3 → a = 8

print(a) # Output: 8

# Bitwise Operators

Bitwise operators perform operations on binary representations of integers.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Bitwise AND | 5 & 3 = 1 |
| ` | ` | Bitwise OR |
| ^ | Bitwise XOR | 5 ^ 3 = 6 |
| ~ | Bitwise NOT | ~5 = -6 |
| << | Left shift | 5 << 1 = 10 |
| >> | Right shift | 5 >> 1 = 2 |

Example:

a = 5 # 0101 in binary

b = 3 # 0011 in binary

print(a & b) # 1 (0001 in binary) print(a | b) # 7 (0111 in binary)

# Identity Operators

Identity operators check whether two variables refer to the same object in memory.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| is | True if both variables point to the same object | a is b |
| is not | True if both variables do not point to the same object | a is not b |

Example:

print(a is b) # True (both point to the same object)

print(a is c) # False (different objects, even though values are the same)

|  |  |  |  |
| --- | --- | --- | --- |
| a = | [1, | 2, | 3] |
| b =  c = | a  [1, | 2, | 3] |

# Membership Operators

Membership operators test for membership in a sequence, such as strings, lists, or tuples.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| in | True if the value is found in the sequence | 3 in [1, 2, 3] → True |
| not in | True if the value is not found in the sequence | 4 not in [1, 2, 3] → True |

Example:

a = [1, 2, 3]

print(2 in a)

# True

print(4 not in a) # True

# Operator Precedence

Operator precedence determines the order in which operations are performed.

For example, multiplication has a higher precedence than addition. Example:

result = 3 + 5 \* 2 # Multiplication happens first, so result is 13 To control precedence, use parentheses:

result = (3 + 5) \* 2 # Now addition happens first, so result is 16

**Assignment Questions Programs:**

##############################################

#Arthimatic Operators:

a = 10

b = 15

sum = a + b sub = a - b mul = a \* b div = a / b

print("########################################")

print("Arthimatic Operators:") print("Sum of A & B :",sum) print("Subtraction of A & B :",sub) print("Multiplication of A & B :", mul) print("Division of A & B :", div)

##############################################

#Comparision Operators:

a = 10

b = 15

gretarthan = a > b lessthan = a < b

greaterthanequeals = a >= b lessthanequals = a <= b bothnotequal = a != b bothequal = a == b

print("########################################")

print("Comparision Operators:") print("a < b:", lessthan ) print("a > b:",gretarthan) print("a <= b:",lessthanequals)

print("a >= b:",greaterthanequeals) print("a != b:",bothnotequal) print("a == b:",bothequal)

##############################################

#Logical Operators:

a = 10

b = 15

andlogic = a and b orlogic = a or b anotlogic = not b bnotlogic = not a

print("########################################")

print("Logical Operators:") print("a and b:", andlogic ) print("a or b:",orlogic) print("a not b:",anotlogic) print("b not a:",bnotlogic)

##############################################

#Assignment Operators:

a = 10

c = 2

d = 3

e = 4

f = 5

b = a

c += a

d -= a

1. \*= a
2. /= a

print("########################################")

print("Assignment Operators:") print("= b value is:", b ) print("+= c value is:", c ) print("-= d value is:", d ) print("\*= e value is:", e ) print("/= f value is:", f )

##############################################

#Bitwise Operators

a = 12

b = 10

bitand = a & b bitor = a | b bitxor = a ^ b

bitleftshift = a << b

bitrightshift = a >> b

print("########################################")

print("Bitwise Operators:") print("a & b value is:", bitand ) print("a | b value is:", bitor ) print("a ^ b value is:", bitxor )

print("a << b value is:", bitleftshift ) print("a >> b value is:", bitrightshift )

##############################################

#Identity Operators:

a = [1,2,3,4,5]

b = a

c = 10

identis = b is a identisnot = c is not a identis1 = a is b identisnot1 = a is not b

print("########################################")

print("Identity Operators:") print("b is a value is:", identis )

print("c is not a value is:", identisnot) print("a is b value is:", identis1 ) print("a is not b value is:", identisnot1)

##############################################

#Membership Operators:

a = [1,2,3,4,5]

b = a

c = 10

memberis = b in a memberisnot = c not in a memberis1 = a in b memberisnot1 = a not in b

print("########################################")

print("Membership Operators:") print("b in a value is:", memberis )

print("c in not a value is:", memberisnot)

print("a in b value is:", memberis1 )

print("a in not b value is:", memberisnot1)

##############################################

**Results:**

##############################################

Arthimatic Operators:

Sum of A & B : 25 Subtraction of A & B : -5 Multiplication of A & B : 150

Division of A & B : 0.6666666666666666 ##############################################

Comparision Operators:

a < b: True a > b: False a <= b: True

a >= b: False a != b: True a == b: False

##############################################

Logical Operators:

a and b: 15 a or b: 10

a not b: False

b not a: False ##############################################

Assignment Operators:

= b value is: 10

+= c value is: 12

-= d value is: -7

\*= e value is: 40

/= f value is: 0.5 ##############################################

Bitwise Operators:

1. & b value is: 8 a | b value is: 14 a ^ b value is: 6
2. << b value is: 12288 a >> b value is: 0

##############################################

Identity Operators:

1. is a value is: True
2. is not a value is: True a is b value is: True
3. is not b value is: False ##############################################

Membership Operators:

1. in a value is: False
2. in not a value is: True a in b value is: False
3. in not b value is: True

**Condition Handlings**

Conditional statements in Python allow you to execute certain code based on the evaluation of a condition. These statements include if, else, and elif (else if). They enable decision-making in your program, where different actions can be taken based on the given conditions.

* If
* else
* elseif
* nestedif

1. **if Statement**

The if statement evaluates a condition and executes the indented block of code if the condition is True.

Syntax:

if condition:

Example:

x = 10

if x > 5:

print("x is greater than 5") # This will be printed since the condition is True

1. **else Statement**

The else statement follows an if statement and is executed when the if condition is False.

Syntax:

if condition:

# Block of code if condition is true

else:

# Block of code if condition is false

Example:

x = 3

if x > 5:

print("x is greater than 5") else:

print("x is 5 or less") # This will be printed

1. **elif (else if) Statement**

The elif statement allows you to check multiple expressions for True and execute the corresponding block for the first True condition. It's used when there are more than two possible outcomes.

Syntax:

Example:

if condition1:

# Block of code if condition1 is true elif condition2:

# Block of code if condition2 is true else:

# Block of code if none of the conditions are true

x = 8

if x > 10:

print("x is greater than 10") elif x == 8:

print("x is 8") # This will be printed else:

print("x is less than 8")

# Nested if Statements

nested if statements, which means using an if or elif inside another if or else block. This is useful when you have complex conditions to evaluate.

Example:

x = 10

y = 5

if x > 5:

if y < 10:

print("x is greater than 5 and y is less than 10") # This will be

printed

**Example of Complex Conditions:**

x = 10

y = 20

if x > y:

print("x is greater than y") elif x == y:

print("x is equal to y") else:

print("x is less than y") # This will be printed since x < y

**List in Python:**

**List & Tuples**

A **list** in Python is a mutable, ordered sequence of elements. Lists can contain elements of different types, and their elements can be modified after creation.

# Characteristics of a List:

* **Mutable**: Elements can be changed, added, or removed.
* **Ordered**: Items have a defined order, and this order will not change unless explicitly modified.
* **Allows duplicates**: Lists can have multiple items with the same value.

Creating a List in Python:

my\_list = [1, 2, 3, 4, 5]

mixed\_list = [1, "Hello", 3.14, True]

Accessing Elements in a List:

You can access elements by their index. The index starts at 0. print(my\_list[0]) # Output: 1 print(mixed\_list[1]) # Output: Hello

# List Methods:

* **append()**: Add an item to the end of the list.
* **insert()***: Insert an item at a specified index.*
* **remove()***: Remove the first item with the specified value.*
* **pop()***: Remove an item at a specified index (or the last item if no index is provided).*
* **sort()***: Sort the list in place.*

Example:

my\_list = [1,2,3,4,5,6,7,8,9]

print("My Original List") print(my\_list)

print("Changing an element of 0th Position") my\_list[0] = 10

print(my\_list) # Output: [10, 2, 3, 4, 5]

print("Adding elements")

my\_list.append(16) # Adds 6 to the end of the list print(my\_list)

print("Inserts elements")

my\_list.insert(2, 100) # Inserts 100 at index 2 print(my\_list)

print("Removing elements the first occurences of 3") my\_list.remove(3) # Removes the first occurrence of 3 print(my\_list)

print("Removes the last element") my\_list.pop() # Removes the last element print(my\_list)

print("Sorting the last element") my\_list.sort()

print(my\_list)

Results:

My Original List

[1, 2, 3, 4, 5, 6, 7, 8, 9]

Changing an element of 0th Position [10, 2, 3, 4, 5, 6, 7, 8, 9]

Adding elements

[10, 2, 3, 4, 5, 6, 7, 8, 9, 16]

Inserts elements

[10, 2, 100, 3, 4, 5, 6, 7, 8, 9, 16]

Removing elements the first occurrences of 3 [10, 2, 100, 4, 5, 6, 7, 8, 9, 16]

Removes the last element

[10, 2, 100, 4, 5, 6, 7, 8, 9]

Sorting the last element

[2, 4, 5, 6, 7, 8, 9, 10, 100]

# Tuples:

A **tuple** in Python is an immutable, ordered sequence of elements. Like lists, tuples can contain elements of different types, but their elements cannot be modified once the tuple is created.

Characteristics of a Tuple:

* **Immutable**: Elements cannot be changed after the tuple is created.
* **Ordered**: Items have a defined order.
* **Allows duplicates**: Tuples can contain multiple items with the same value. Creating Tuple in Python:

my\_tuple = (1, 2, 3)

mixed\_tuple = (1, "Hello", 3.14, True)

# A tuple with one element needs a trailing comma single\_element\_tuple = (5,)

Accessing Elements in a Tuple:

Like lists, you can access tuple elements by their index.

print(my\_tuple[0]) # Output: 1

Methods in Tuples:

Since tuples are immutable, they have fewer methods than lists. However, two useful methods are:

* **count()**: Returns the number of times a specified value occurs in a tuple.
* **index()**: Returns the index of the first occurrence of a specified value. Example:

my\_tuple = (1, 22, 3, 2, 4) print(my\_tuple.count(1)) # Output: 2 print(my\_tuple.index(3)) # Output: 4

Uses of Tuples Instead of Lists?

* **Immutability**: Tuples are immutable, which means they are safer when you want to ensure that data does not change accidentally.
* **Performance**: Tuples are slightly more efficient in terms of memory and performance compared to lists.
* **Used as keys in dictionaries**: Tuples can be used as dictionary keys because they are hash-able, while lists cannot.

# Key Differences Between Lists and Tuples:

|  |  |  |
| --- | --- | --- |
| **Feature** | **List** | **Tuple** |
| **Mutability** | Mutable (can be changed) | Immutable (cannot be changed) |
| **Syntax** | Created with [] | Created with () |
| **Methods** | More methods (e.g., append(), remove()) | Fewer methods (e.g., count(), index()) |
| **Performance** | Slower, uses more memory | Faster, uses less memory |
| **Use Cases** | Use when data can change | Use when data should remain constant |

**Basic Questions:**

Q1: What is a list in Python, and how is it used in DevOps?

* List is collection of elements
* List is mutable in python.
* Sequence of elements in the define list.
* List can be multiple data types.
* In devops list can be used to store and manipulate data, such as configurations, target servers, deployment.

Q2: How do you create a list in Python, and can you provide an example related to DevOps?

* List can be define in the square brackets – []
* Example: my\_list = [1,2,3,4]
* In devops we will use list to define the instance type, list of S3 buckets etc.

Q3: What is the difference between a list and a tuple in Python, and when would you choose one over the other in a DevOps context?

* List is mutable and tuple is immutable
* Syntax for list is [] and for tuple is ()
* Performance wise list is slower, uses more memory, where tuple is less and uses less memory.
* For the server configuration, target servers and deployment nodes we can use the list.
* For admin related information, deployment steps and configuration servers stores we will use tuples.

Q4: How can you access elements in a list, and provide a DevOps-related example?

- My\_list = [1,2,3,4,5]

* Access the list using the index – my\_list[0]  1

Q5: How do you add an element to the end of a list in Python? Provide a DevOps example.

- My\_list = [1,2,3,4]

* My\_list.append(5)

Q6: How can you remove an element from a list in Python, and can you provide a DevOps use case?

- My\_list = [1,2,3,4]

* My\_list.remove(2)

# Loops

Loops in Python allow you to execute a block of code repeatedly, either for a specified number of times or while a certain condition is true. Python provides two main types of loops: for loops and while loops.

**for** Loop:

A for loop in Python is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each element in the sequence.

Syntax:

for variable in sequence:

# Code to execute in each iteration

Example:

fruits = ["apple", "banana", "cherry"] for fruit in fruits:

print(fruit)

Output:

apple banana cherry

Using **range()** with for Loop:

The range() function generates a sequence of numbers, which is commonly used with loops.

Example:

for i in range(4): print(i)

Output:

0

1

2

3

**while** Loop

A while loop repeats as long as a specified condition is True. It is useful when the number of iterations is not predetermined.

Syntax:

while condition:

# Code to execute while the condition is true

Example:

i = 1

while i <= 4:

print(i)

i += 1 # Increment the counter to avoid an infinite loop

Output:

1

2

3

4

**break** and **continue** Statements

* The break statement is used to exit the loop prematurely, even if the condition for the loop has not become False.
* The continue statement skips the current iteration and proceeds to the next iteration of the loop.

Example for Break:

for i in range(5): if i == 3:

break # Exits the loop when i equals 3 print(i)

Output:

0

1

2

Example for Continue:

for i in range(5): if i == 3:

continue # Skips the iteration when i equals 3 print(i)

Output:

0

1

2

4

else in Loops:

Python allows you to use an else block with for and while loops. The else block is executed when the loop completes normally, i.e., without hitting a break statement.

Example with for loop:

for i in range(5): print(i)

else:

print("Loop completed")

Output:

0

1

2

3

4

Loop completed

Example with while loop:

i = 1

while i < 5:

print(i) i += 1

else:

print("While loop completed")

Output:

1

2

3

4

While loop completed

**Nested** Loops

You can nest loops, meaning you can place one loop inside another loop. Each iteration of the outer loop runs the inner loop to completion.

Example:

for i in range(3):

for j in range(2):

print(f"i = {i}, j = {j}")

Output:

i = 0, j = 0

i = 0, j = 1

i = 1, j = 0

i = 1, j = 1

i = 2, j = 0

i = 2, j = 1

Loop Control with **pass**

The pass statement in Python is used when you need a statement syntactically but do not want to execute any code. It is often used as a placeholder.

Example:

for i in range(5): if i == 3:

pass # Placeholder for future code print(i)

Output:

1

2

3

4

# Working with Lists

We will deep dive into the list exercise examples to understand more on this.

In Python, you can take input from the user while running the program using the **input()** function. This function waits for the user to type something and press Enter, and then it returns the input as a string.

Example:

name = input("Enter your name: ") print("Hello, {name}!")

Output:

Enter your name: Sajid Hello, Sajid!

Taking Numerical Input:

By default, the input() function returns the data as a string. If you want to take a numerical input, you need to convert the string to an integer (int()) or a floating-point number (float()).

* Example for Integer Input:

age = int(input("Enter your age: ")) print("You are {age} years old.")

* Example for Float Input:

height = float(input("Enter your height in meters: ")) print("Your height is {height} meters.")

Handling Multiple Inputs

If you want to take multiple inputs in a single line, you can use the **split()** method to divide the input into multiple values.

* Example:

x, y = input("Enter two numbers separated by space: ").split() print("x = {x}, y = {y}")

Output:

Enter two numbers separated by space: 5 10 x = 5, y = 10

* You can also directly convert the input values to integers or floats: x, y = map(int, input("Enter two integers: ").split()) print("Sum: {x + y}")

Taking Input in a Loop

You can use input() inside a loop to keep asking for input until a certain condition

is met.

* Example with while Loop:

while True:

data = input("Enter something (or type 'exit' to quit): ") if data.lower() == "exit":

break

print("You entered:”, data)

In this example, the program keeps asking for input until the user types "exit", after which the loop terminates.

* Example:

Full Program to Perform Addition Based on User Input

# Python program to take two numbers as input and print their sum num1 = float(input("Enter first number: "))

num2 = float(input("Enter second number: ")) # Calculate and display sum

sum\_of\_numbers = num1 + num2

print("The sum of {num1} and {num2} is {sum\_of\_numbers}")

Output:

Enter first number: 5.5 Enter second number: 4.5

The sum of 5.5 and 4.5 is 10.0

Exception Handling:

Python provides the try, except, else, and finally blocks to handle exceptions gracefully without crashing the program.

Syntax:

try:

# Code that might raise an exception except ExceptionType:

# Code that runs if the exception occurs

Basic Example: Handling an Exception:

try:

result = 10 / 0 except ZeroDivisionError:

print("Cannot divide by zero!")

Output:

Cannot divide by zero!

**Example**: Listing the files and folders from the user input folder. Steps:

* Taking the user input using input()
* Listing the files & folders using the for loop
* Using OS module
* Print the output
* Handling the known errors using try & except Final Program:

import os

def listdirectory():

path\_name = input("Enter the path:").split(' ') print(path\_name)

for paths in path\_name: try:

files = os.listdir(paths) except FileNotFoundError:

print("Enter Valid Folder Path") continue

#for path in paths: #print(path) #print(files)

for file in files: print(file)

listdirectory()

# Dictionaries & Sets

A dictionary in Python is a data structure that allows you to store and retrieve values using key-value pairs. Each key in the dictionary must be unique and immutable (such as strings, numbers, or tuples), while values can be of any data type and can be duplicated.

Syntax:

my\_dict = {

"key1": "value1",

"key2": "value2",

"key3": "value3"

}

# Creating a Dictionary

# Dictionary of fruits and their colors fruit\_colors = {

"apple": "red",

"banana": "yellow",

"cherry": "red"

}

# Accessing Dictionary Elements

You can access the value of a specific key using square brackets [].

print(fruit\_colors["apple"]) # Output: red

You can also use the get() method, which returns None if the key doesn't exist instead of raising an error.

print(fruit\_colors.get("grape")) # Output: None

# Adding and Updating Elements

To add or update an entry, simply assign a value to a key.

# Adding a new entry fruit\_colors["grape"] = "purple"

# Updating an existing entry fruit\_colors["banana"] = "green"

print(fruit\_colors) # Output: {'apple': 'red', 'banana': 'green', 'cherry': 'red', 'grape': 'purple'}

# Removing Elements

You can remove a key-value pair using the del keyword or the pop() method.

# Using del

del fruit\_colors["cherry"]

# Using pop fruit\_colors.pop("banana")

print(fruit\_colors) # Output: {'apple': 'red', 'grape': 'purple'}

# Example using Loops:

# Dictionary of fruits and their colors fruit\_colors = {

"apple": "red",

"banana": "yellow",

"cherry": "red"

}

# Looping through keys

for fruit in fruit\_colors:

print(fruit) # Output: apple, grape

# Looping through values

for color in fruit\_colors.values(): print(color) # Output: red, purple

# Looping through key-value pairs

for fruit, color in fruit\_colors.items(): print("{fruit} is {color}")

**Output:**

apple banana cherry red yellow red

apple is red banana is yellow cherry is red

# Adding and Removing Elements

* **add()**: Adds a new element to the set.
* **remove()**: Removes an element, raises an error if the element is not found.
* **discard()**: Removes an element, does not raise an error if the element is not found.
* **pop()**: Removes and returns an arbitrary element.
* **clear()**: Removes all elements from the set.

Example:

# Adding an element my\_set.add(5)

# Removing an element my\_set.remove(3)

# Removing an element that might not exist my\_set.discard(10)

# Set:

print(my\_set) # Output: {1, 2, 4, 5}

Unordered collections of unique elements. Sets support mathematical set

operations like union, intersection, and difference.

Example:

set\_a = {1, 2, 3, 4}

set\_b = {3, 4, 5, 6}

# Union (all unique elements from both sets) print(set\_a | set\_b) # Output: {1, 2, 3, 4, 5, 6}

# Intersection (common elements) print(set\_a & set\_b) # Output: {3, 4}

# Difference (elements in set\_a but not in set\_b) print(set\_a - set\_b) # Output: {1, 2}

# Symmetric Difference (elements in either set\_a or set\_b, but not both) print(set\_a ^ set\_b) # Output: {1, 2, 5, 6}

Example Use Case:

- Get Pull Request creators & count information from GIT Repo.

# Program to demonstrate integration with GitHub to fetch the

# details of Users who created Pull requests(Active) on Kubernetes Github repo.

import requests

# URL to fetch pull requests from the GitHub API

url = f'https://api.github.com/repos/mdsajid786/Python/pulls'

# Make a GET request to fetch pull requests data from the GitHub API response = requests.get(url) # Add headers=headers inside get() for authentication

# Only if the response is successful if response.status\_code == 200:

# Convert the JSON response to a dictionary pull\_requests = response.json()

# Create an empty dictionary to store PR creators and their counts pr\_creators = {}

# Iterate through each pull request and extract the creator's name for pull in pull\_requests:

creator = pull['user']['login'] if creator in pr\_creators:

pr\_creators[creator] += 1 else:

pr\_creators[creator] = 1

# Display the dictionary of PR creators and their counts print("PR Creators and Counts:")

for creator, count in pr\_creators.items(): print(f"{creator}: {count} PR(s)")

else:

print(f"Failed to fetch data. Status code: {response.status\_code}")

Output:

PR Creators and Counts: DeepTests1: 1 PR(s)

SudheerT: 1 PR(s)

AparnaN: 1 PR(s)

ShubhangiS: 1 PR(s)

# File Operations

Python provides built-in functions for file handling, allowing you to create, read, write, and delete files. File operations are done using the open() function, which allows you to work with files in different modes, such as reading (r), writing (w), and appending (a).

Basic File Handling Workflow:

* Open a file using open().
* Read or write to the file.
* Close the file using close() to free up system resources.

# Opening a File

The open() function takes two arguments: the filename and the mode. The available modes are:

* "r": Read (default mode). Opens a file for reading, raises an error if the file does not exist.
* "w": Write. Opens a file for writing (creates a new file or overwrites the existing content).
* "a": Append. Opens a file for appending (adds new content without overwriting).
* "r+": Read and write.
* "w+": Write and read (overwrites the existing content).
* "a+": Append and read.

Example:

file = open("example.txt", "w") # Open file for writing file.write("Hello, world!") # Write to the file file.close() # Close the file

# Reading a File

You can read a file using the read(), readline(), or readlines() methods.

* **read()**: Reads the entire file.
* **readline()**: Reads one line at a time.
* **readlines()**: Reads all lines and returns a list of strings.

Example: Reading a File

file = open("example.txt", "r")

content = file.read() # Read entire file print(content)

file.close()

Example: Reading Line by Line

file = open("example.txt", "r")

line = file.readline() # Read one line while line:

print(line.strip()) # Strip to remove trailing newline line = file.readline()

file.close()

# Writing to a File

When writing to a file, you can use the write() method. If the file doesn't exist, it will be created. If it does exist, the content will be overwritten in "w" mode.

Example:

file = open("example.txt", "w") file.write("This is a new line of text.") file.close()

# Appending to a File

When you want to add data to the end of an existing file without overwriting its content, use the "a" mode.

Example:

file = open("example.txt", "a") file.write("\nAppending a new line.") file.close()

# Deleting a File

To delete a file, use the os module.

Example:

import os

if os.path.exists("example.txt"): os.remove("example.txt")

else:

print("The file does not exist")

# File Methods Overview

* **open()**: Opens a file for reading or writing.
* **read()**: Reads the entire file content.
* **readline()**: Reads a single line from the file.
* **readlines()**: Reads all lines from the file and returns them as a list.
* **write()**: Writes content to the file.
* **writelines()**: Writes a list of strings to the file.
* **close()**: Closes the file.

Example use case:

Updating server config path value:

* This script opens the `server.conf` file in read mode and stores its content in a list.

It then reopens the file in write mode, searches for the specified keyword provided in the function definition, and updates the corresponding values based on the input given in the function.

def update\_server\_config(file\_path, key, value):

# Read the existing content of the server configuration file with open(file\_path, 'r') as file:

lines = file.readlines()

# Update the configuration value for the specified key with open(file\_path, 'w') as file:

for line in lines:

# Check if the line starts with the specified key if key in line:

# Update the line with the new value file.write(key + "=" + value + "\n")

else:

# Keep the existing line as it is file.write(line)

# Path to the server configuration file server\_config\_file = 'server.conf'

# Key and new value for updating the server configuration key\_to\_update = 'MAX\_CONNECTIONS'

new\_value = '600' # New maximum connections allowed

# Update the server configuration file update\_server\_config(server\_config\_file, key\_to\_update, new\_value)

# Boto3 Overview

* Boto3 is the Amazon Web Services (AWS) Software Development Kit (SDK) for Python.
* It allows Python developers to write code that interacts with various AWS services such as S3, EC2, DynamoDB, and more.
* Boto3 provides both low-level service access and higher-level resource abstractions to make it easier to work with AWS.

# Installing Boto3

Before using Boto3, you need to install it via pip:

pip install boto3

# Configuring Boto3

Boto3 can be configured using the AWS CLI credentials, or by manually specifying credentials in the code.

AWS CLI Configuration

If you’ve installed and configured the AWS CLI, Boto3 will automatically use those credentials. You can configure the AWS CLI by running:

aws configure

This will prompt you to provide:

* AWS Access Key ID
* AWS Secret Access Key
* Default Region Name
* Default Output Format (e.g., json)

# Session, Client, and Resource

* **Client**: Provides low-level access to AWS services. Every operation requires more manual handling, and the response is returned in a dictionary format.

s3 = boto3.client('s3')

* **Resource**: Provides higher-level, object-oriented abstractions. It’s easier to use

for common tasks but not available for all AWS services.

s3 = boto3.resource('s3')

* **Session**: Allows you to manage multiple AWS configurations.

session = boto3.Session(aws\_access\_key\_id='KEY', aws\_secret\_access\_key='SECRET')

s3 = session.client('s3')

Example: Uploading Files to S3

* creating a new S3 bucket, uploading a file to it, and then listing the files in the bucket:

import boto3

# Create an S3 client s3 = boto3.client('s3')

# Create a new bucket s3.create\_bucket(Bucket='my-new-bucket')

# Upload a file to the bucket

s3.upload\_file('local\_file.txt', 'my-new-bucket', 'file\_in\_s3.txt')

# List files in the bucket

response = s3.list\_objects\_v2(Bucket='my-new-bucket') for obj in response.get('Contents', []):

print(obj['Key'])

In DevOps world we can use this to create the AWS resources, Cost Optimizations etc.

Use Case:

# JIRA & GitHub Integration

Creating automatic JIRA ticket from GitHub. Steps to be followed:

* Create JIRA login
* JIRA API Calls
* Create the GitHub workflow to integrate with JIRA
* Run the python Application List JIRA Projects Script:

# This code sample uses the 'requests' library: # [http://docs.python-requests.org](http://docs.python-requests.org/)

import requests

from requests.auth import HTTPBasicAuth import json

url = "https://mdsajid020.atlassian.net/rest/api/3/project" API\_TOKEN="ajljagpoelgaldsjglajljaljsdglajdlfgjaljlsdjbvlajbl" auth = HTTPBasicAuth("[mdsajid020@gmail.com](mailto:mdsajid020@gmail.com)", API\_TOKEN)

headers = {

"Accept": "application/json"

}

response = requests.request( "GET",

url, headers=headers, auth=auth

)

output = json.loads(response.text)

name = output[0]["name"] print(name)

Create JIRA Script:

# This code sample uses the 'requests' library: # [http://docs.python-requests.org](http://docs.python-requests.org/)

import requests

from requests.auth import HTTPBasicAuth import json

url = "https://mdsajid020.atlassian.net/rest/api/3/issue" API\_TOKEN = "ATATT3xFfGF0FEN7BU7iVpTd3sIa7AOUXX18wWO"

auth = HTTPBasicAuth(["mdsajid020@gmail.com](mailto:mdsajid020@gmail.com)", API\_TOKEN)

headers = {

"Accept": "application/json", "Content-Type": "application/json"

}

payload = json.dumps( { "fields": {

"description": { "content": [

{

"content": [

{

"text": "My first jira ticket", "type": "text"

}

],

"type": "paragraph"

}

],

"type": "doc", "version": 1

},

"project": {

"id": "10000"

},

"issuetype": {

"id": "10001"

},

"summary": "First JIRA Ticket",

},

"update": {}

} )

response = requests.request( "POST",

url, data=payload, headers=headers, auth=auth

)

print(json.dumps(json.loads(response.text), sort\_keys=True, indent=4, separators=(",", ": ")))

Final Python App using flask:

# This code sample uses the 'requests' library: # [http://docs.python-requests.org](http://docs.python-requests.org/)

import requests

from requests.auth import HTTPBasicAuth import json

from flask import Flask app = Flask( name )

# Define a route that handles GET requests @app.route('/createJira', methods=['POST']) def createJira():

url = "https://mdsajid020.atlassian.net/rest/api/3/issue" API\_TOKEN="ATATT3xFfGF0FEN7BU7iVpT"

auth = HTTPBasicAuth(["mdsajid020@gmail.com"](mailto:mdsajid020@gmail.com), API\_TOKEN)

headers = {

"Accept": "application/json", "Content-Type": "application/json"

}

payload = json.dumps( { "fields": { "description": {

"content": [

{

"content": [

{

"text": "Order entry fails when selecting

supplier.",

"type": "text"

}

],

"type": "paragraph"

}

],

"type": "doc", "version": 1

},

"project": {

"id": "10000"

},

"issuetype": {

"id": "10001"

},

"summary": "Main order flow broken",

},

"update": {}

} )

response = requests.request( "POST",

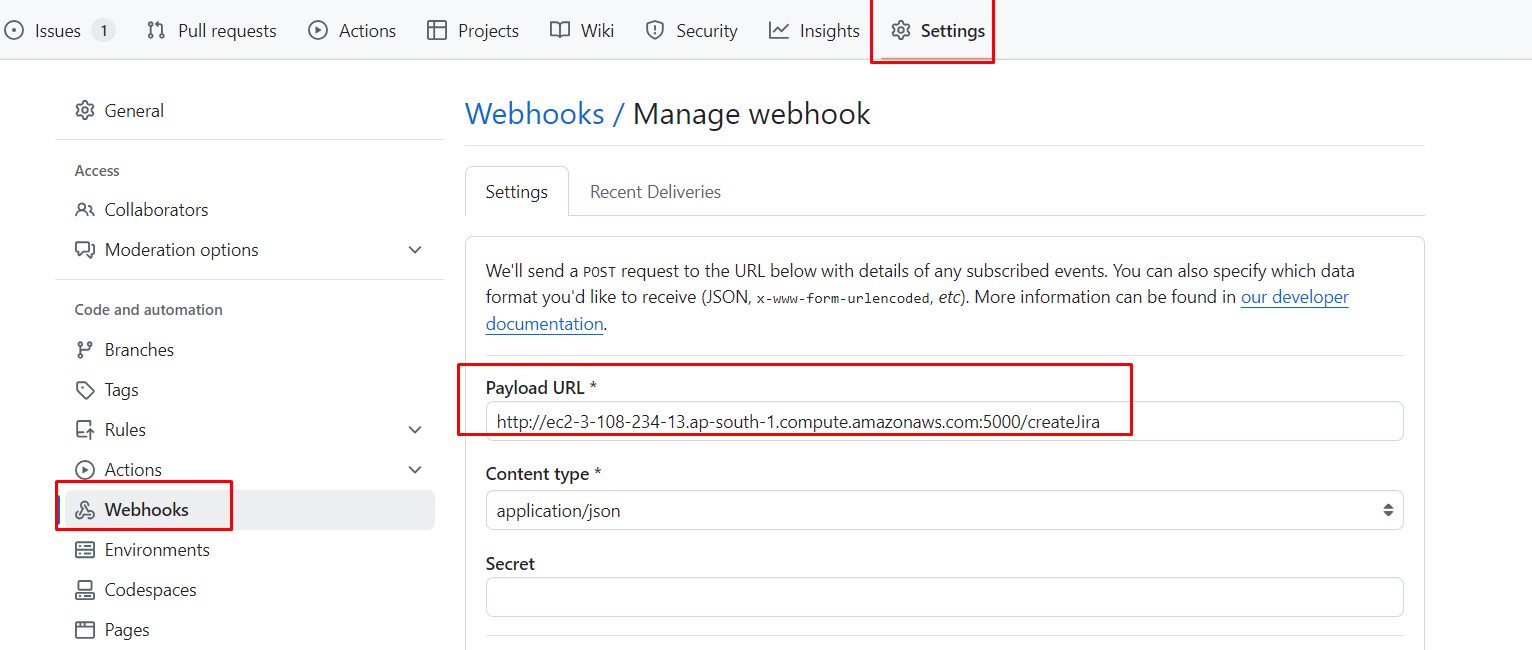
url, data=payload, headers=headers, auth=auth

)

return json.dumps(json.loads(response.text), sort\_keys=True, indent=4, separators=(",", ": "))

if name == ' main ': app.run(host='0.0.0.0', port=5000)

Need to configure this application URL in the GitHub Webhook section. Refer snapshot



# Basic Interview Questions

**Q1. Describe a real-world example of how you used Python to solve a DevOps challenge.**

* Here you can talk about the projects that we did in this series
  + GitHub Webhooks
  + JIRA integration
  + File Operations

# Q2. Discuss the challenges that you faced while using Python for DevOps and how did you overcome it.

* Here you can mention about a challenge that you faced while implementating a Python project for DevOps that we learnt.

# Q3. How can you secure your Python code and scripts?

* Handle any sensetive information using Input variables, command line arguments or env vars.

# Q4. Explain the difference between mutable and immutable objects.

In Python, mutable objects can be altered after creation, while immutable objects cannot be changed once created. For instance:

Mutable objects like lists can be modified:

my\_list = [1, 2, 3]

my\_list[0] = 0 # Modifying an element in the list print(my\_list) # Output: [0, 2, 3]

Immutable objects like tuples cannot be altered:

my\_tuple = (1, 2, 3)

# Attempting to change a tuple will result in an error # my\_tuple[0] = 0

# Q5. Differentiate between list and tuple in Python.

Lists are mutable and typically used for storing collections of items that can be changed, while tuples are immutable and commonly used to store collections of items that shouldn't change. Examples:

List:

my\_list = [1, 2, 3]

my\_list.append(4) # Modifying by adding an element print(my\_list) # Output: [1, 2, 3, 4]

Tuple:

my\_tuple = (1, 2, 3)

# Attempting to modify a tuple will result in an error # my\_tuple.append(4)

# Q6. Explain the use of virtualenv.

Virtualenv creates isolated Python environments, allowing different projects to use different versions of packages without conflicts. Example:

Creating a virtual environment:

Creating a virtual environment named 'myenv' virtualenv myenv

Activating the virtual environment:

On Windows myenv\Scripts\activate On Unix or MacOS

source myenv/bin/activate

# Q7. What are decorators in Python?

Decorators modify the behavior of functions. They take a function as an argument, add some functionality, and return another function without modifying the original function's code. Example:

Defining a simple decorator:

def my\_decorator(func): def wrapper():

print("Something is happening before the function is called.") func()

print("Something is happening after the function is called.")

return wrapper

@my\_decorator def say\_hello():

print("Hello!")

say\_hello()

# Q8. How does exception handling work in Python?

Exception handling in Python uses try, except, else, and finally blocks. Example: Handling division by zero exception:

try:

result = 10 / 0

except ZeroDivisionError:

print("Division by zero is not allowed.") else:

print("Division successful:", result) finally:

print("Execution completed.")

# Q9. What's the difference between append() and extend() for lists?

append() adds a single element to the end of a list, while extend() adds multiple elements by appending elements from an iterable. Example:

Using append():

my\_list = [1, 2, 3] my\_list.append(4)

print(my\_list) # Output: [1, 2, 3, 4] Using extend():

my\_list = [1, 2, 3]

my\_list.extend([4, 5])

print(my\_list) # Output: [1, 2, 3, 4, 5]

# Q10. Explain the use of lambda functions in Python.

Lambda functions are anonymous functions used for short tasks. Example: Defining and using a lambda function:

square = lambda x: x\*\*2

print(square(5)) # Output: 25

# Q11. What are the different types of loops in Python?

Python has for loops and while loops. Example:

Using for loop:

for i in range(5): print(i)

Using while loop:

i = 0

while i < 5: print(i)

i += 1

# Q12. Explain the difference between == and is operators.

The == operator compares the values of two objects, while the is operator checks if two variables point to the same object in memory.

Example:

Using ==:

a = [1, 2, 3]

b = [1, 2, 3]

print(a == b) # Output: True (because values are equal) Using is:

a = [1, 2, 3]

b = a

print(a is b) # Output: True (because they reference the same object)

# Q13. What is the use of the pass keyword?

The pass keyword is a no-operation placeholder used when a statement is syntactically needed but no action is required. Example:

Using pass:

def placeholder\_function():

pass # To be implemented later

# Q14. What is the difference between global and local variables?

Global variables are defined outside functions and can be accessed anywhere in the code, while local variables are defined inside functions and are only accessible within that function's scope. Example:

Using a global variable:

global\_var = 10

def my\_function(): print(global\_var)

my\_function() # Output: 10 Using a local variable:

def my\_function(): local\_var = 5 print(local\_var)

my\_function() # Output: 5

# Attempting to access local\_var outside the function will result in an error

# Q15. Explain the difference between open() and with open() statement.

open() is a built-in function used to open a file and return a file object. However, it's crucial to manually close the file using file\_object.close(). Conversely, with open() is a context manager that automatically handles file closure, ensuring clean-up even if exceptions occur.

Example:

file = open('example.txt', 'r') content = file.read() file.close()

Using with open():

with open('example.txt', 'r') as file: content = file.read()

# File is automatically closed when the block exits