PYTHON NOTES

Certainly! The choice between using shell scripting and Python in DevOps depends on the specific task or problem you're trying to solve. Both have their strengths and are suitable for different scenarios. Here are some guidelines to help you decide when to use each:

Use Shell Scripting When:

1. **System Administration Tasks:** Shell scripting is excellent for automating routine system administration tasks like managing files, directories, and processes. You can use shell scripts for tasks like starting/stopping services, managing users, and basic file manipulation.
2. **Command Line Interactions:** If your task primarily involves running command line tools and utilities, shell scripting can be more efficient. It's easy to call and control these utilities from a shell script.
3. **Rapid Prototyping:** If you need to quickly prototype a solution or perform one-off tasks, shell scripting is usually faster to write and execute. It's great for ad-hoc tasks.
4. **Text Processing:** Shell scripting is well-suited for tasks that involve text manipulation, such as parsing log files, searching and replacing text, or extracting data from text-based sources.
5. **Environment Variables and Configuration:** Shell scripts are useful for managing environment variables and configuring your system.

Use Python When:

1. **Complex Logic:** Python is a full-fledged programming language and is well-suited for tasks that involve complex logic, data structures, and algorithms. If your task requires extensive data manipulation, Python can be a more powerful choice.
2. **Cross-Platform Compatibility:** Python is more platform-independent than shell scripting, making it a better choice for tasks that need to run on different operating systems.
3. **API Integration:** Python has extensive libraries and modules for interacting with APIs, databases, and web services. If your task involves working with APIs, Python may be a better choice.
4. **Reusable Code:** If you plan to reuse your code or build larger applications, Python's structure and modularity make it easier to manage and maintain.
5. **Error Handling:** Python provides better error handling and debugging capabilities, which can be valuable in DevOps where reliability is crucial.
6. **Advanced Data Processing:** If your task involves advanced data processing, data analysis, or machine learning, Python's rich ecosystem of libraries (e.g., Pandas, NumPy, SciPy) makes it a more suitable choice.

Data Types

In programming, a data type is a classification or categorization that specifies which type of value a variable can hold. Data types are essential because they determine how data is stored in memory and what operations can be performed on that data. Python, like many programming languages, supports several built-in data types. Here are some of the common data types in Python:

1. Numeric Data Types:

- **int:** Represents integers (whole numbers). Example: `x = 5`

- **float:** Represents floating-point numbers (numbers with decimal points). Example: `y = 3.14`

- **complex:** Represents complex numbers. Example: `z = 2 + 3j`

2. Sequence Types:

- **str:** Represents strings (sequences of characters). Example: `text = "Hello, World"`

- **list:** Represents lists (ordered, mutable sequences). Example: `my_list = [1, 2, 3]`

- **tuple:** Represents tuples (ordered, immutable sequences). Example: `my_tuple = (1, 2, 3)`

3. Mapping Type:

- **dict:** Represents dictionaries (key-value pairs). Example: `my_dict = {'name': 'John', 'age': 30}`

4. Set Types:

- **set:** Represents sets (unordered collections of unique elements). Example: `my_set = {1, 2, 3}`

- **frozenset:** Represents immutable sets. Example: `my_frozenset = frozenset([1, 2, 3])`

5. Boolean Type:

- **bool:** Represents Boolean values (`True` or `False`). Example: `is_valid = True`

6. Binary Types:

- **bytes:** Represents immutable sequences of bytes. Example: `data = b'Hello'`

- **bytearray:** Represents mutable sequences of bytes. Example: `data = bytearray(b'Hello')`

7. None Type:

- **NoneType:** Represents the `None` object, which is used to indicate the absence of a value or a null value.

8. Custom Data Types:

- You can also define your custom data types using classes and objects.

Strings

1. String Data Type in Python:

- In Python, a string is a sequence of characters, enclosed within single (' '), double (" "), or triple (''' ''' or """ """) quotes.
- Strings are immutable, meaning you cannot change the characters within a string directly. Instead, you create new strings.
- You can access individual characters in a string using indexing, e.g., my_string[0] will give you the first character.

- Strings support various built-in methods, such as len(), upper(), lower(), strip(), replace(), and more, for manipulation.

2. String Manipulation and Formatting:

- Concatenation: You can combine strings using the + operator.
- Substrings: Use slicing to extract portions of a string, e.g., my_string[2:5] will extract characters from the 2nd to the 4th position.
- String interpolation: Python supports various ways to format strings, including f-strings (f"...{variable}..."), %-formatting ("%s %d" % ("string", 42)), and str.format().
- Escape sequences: Special characters like newline (\n), tab (\t), and others are represented using escape sequences.
- String methods: Python provides many built-in methods for string manipulation, such as split(), join(), and startswith().

Numeric Data Type

1. Numeric Data Types in Python (int, float):

- Python supports two primary numeric data types: int for integers and float for floating-point numbers.
- Integers are whole numbers, and floats can represent both whole and fractional numbers.
- You can perform arithmetic operations on these types, including addition, subtraction, multiplication, division, and more.
- Be aware of potential issues with floating-point precision, which can lead to small inaccuracies in calculations.
- Python also provides built-in functions for mathematical operations, such as abs(), round(), and math module for advanced functions.

Regex

1. Regular Expressions for Text Processing:

- Regular expressions (regex or regexp) are a powerful tool for pattern matching and text processing.
- The re module in Python is used for working with regular expressions.
- Common metacharacters: . (any character), * (zero or more), + (one or more), ? (zero or one), [] (character class), | (OR), ^ (start of a line), $ (end of a line), etc.
- Examples of regex usage: matching emails, phone numbers, or extracting data from text.
- re module functions include re.match(), re.search(), re.findall(), and re.sub() for pattern matching and replacement.

Examples for Strings:-

1. Concat :

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)
```

2. Length:

```
text = "Python is awesome"
length = len(text)
print("Length of the string:", length)
```

3. LowerCase :

```
text = "Python is awesome"
uppercase = text.upper()
lowercase = text.lower()
print("Uppercase:", uppercase)
print("Lowercase:", lowercase)
```

4. Replace :

```
text = "Python is awesome"
new_text = text.replace("awesome", "great")
print("Modified text:", new_text)
```

5. Split :

```
text = "Python is awesome"
words = text.split()
print("Words:", words)
```

6. Strip :

```
text = "   Some spaces around   "
stripped_text = text.strip()
print("Stripped text:", stripped_text)
```

7. Substring :

```
text = "Python is awesome"
substring = "is"
if substring in text:
    print(substring, "found in the text")
```

Float :--

```
# Float variables

num1 = 5.0
num2 = 2.5

# Basic Arithmetic

result1 = num1 + num2
print("Addition:", result1)
result2 = num1 - num2
print("Subtraction:", result2)
result3 = num1 * num2
print("Multiplication:", result3)
result4 = num1 / num2
print("Division:", result4)

# Rounding

result5 = round(3.14159265359, 2)  # Rounds to 2 decimal places
print("Rounded:", result5)
```

Int :--

```
# 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)
```

Regex :--

findall :

```
import re

text = "The quick brown fox"
```

```
pattern = r"brown"
search = re.search(pattern, text)
if search:
    print("Pattern found:", search.group())
else:
    print("Pattern not found")
```

match:

```
import re

text = "The quick brown fox"
pattern = r"quick"
match = re.match(pattern, text)
if match:
    print("Match found:", match.group())
else:
    print("No match")
```

replace:

```
import re

text = "The quick brown fox jumps over the lazy brown dog"
pattern = r"brown"
replacement = "red"
new_text = re.sub(pattern, replacement, text)
print("Modified text:", new_text)
```

search :

```
import re

text = "The quick brown fox"
pattern = r"brown"
search = re.search(pattern, text)
if search:
    print("Pattern found:", search.group())
else:
    print("Pattern not found")
```

split :

```
import re

text = "apple,banana,orange,grape"
pattern = r","
split_result = re.split(pattern, text)
print("Split result:", split_result)
```

Keywords in Python:

Keywords are reserved words in Python that have predefined meanings and cannot be used as variable names or identifiers. These words are used to define the structure and logic of the program. They are an integral part of the Python language and are case-sensitive, which means you must use them exactly as specified.

Here are some important Python keywords:

1. and: It is a logical operator that returns True if both operands are true.
2. or: It is a logical operator that returns True if at least one of the operands is true.
3. not: It is a logical operator that returns the opposite of the operand's truth value.
4. if: It is used to start a conditional statement and is followed by a condition that determines whether the code block is executed.
5. else: It is used in conjunction with if to define an alternative code block to execute when the if condition is False.
6. elif: Short for "else if," it is used to check additional conditions after an if statement and is used in combination with if and else.
7. while: It is used to create a loop that repeatedly executes a block of code as long as a specified condition is true.
8. for: It is used to create a loop that iterates over a sequence (such as a list, tuple, or string) and executes a block of code for each item in the sequence.
9. in: Used with for, it checks if a value is present in a sequence.
10. try: It is the beginning of a block of code that is subject to exception handling. It is followed by except to catch and handle exceptions.
11. except: Used with try, it defines a block of code to execute when an exception is raised in the corresponding try block.
12. finally: Used with try, it defines a block of code that is always executed, whether an exception is raised or not.
13. def: It is used to define a function in Python.
14. return: It is used within a function to specify the value that the function should return.
15. class: It is used to define a class, which is a blueprint for creating objects in object-oriented programming.
16. import: It is used to import modules or libraries to access their functions, classes, or variables.
17. from: Used with import to specify which specific components from a module should be imported.
18. as: Used with import to create an alias for a module, making it easier to reference in the code.

19. True: It represents a boolean value for "true."
20. False: It represents a boolean value for "false."
21. None: It represents a special null value or absence of value.
22. is: It is used for identity comparison, checking if two variables refer to the same object in memory.
23. lambda: It is used to create small, anonymous functions (lambda functions).
24. with: It is used for context management, ensuring that certain operations are performed before and after a block of code.
25. global: It is used to declare a global variable within a function's scope.
26. nonlocal: It is used to declare a variable as nonlocal, which allows modifying a variable in an enclosing (but non-global) scope.

Understanding Variables in Python:

late it, and make our code more flexible and reusable.

Example:

```
# Assigning a value to a variable

my_variable = 42
# Accessing the value of a variable
print(my_variable)     # Output: 42
```

Variable Scope and Lifetime:

Variable Scope: In Python, variables have different scopes, which determine where in the code the variable can be accessed. There are mainly two types of variable scopes:

1. **Local Scope:** Variables defined within a function have local scope and are only accessible inside that function.

```
def my_function():
      x = 10  # Local variable
   print(x)
my_function()
print(x)   # This will raise an error since 'x' is not defined outside the function.
```

2. **Global Scope:** Variables defined outside of any function have global scope and can be accessed throughout the entire code.

```
y = 20   # Global variable
def another_function():
          print(y)     # This will access the global variable 'y'
another_function()
print(y)      # This will print 20
```

Variable Lifetime: The lifetime of a variable is determined by when it is created and when it is destroyed or goes out of scope. Local variables exist only while the function is being executed, while global variables exist for the entire duration of the program.

Variable Naming Conventions and Best Practices:

It's important to follow naming conventions and best practices for variables to write clean and maintainable code:

- Variable names should be descriptive and indicate their purpose.
- Use lowercase letters and separate words with underscores (snake_case) for variable names.
- Avoid using reserved words (keywords) for variable names.
- Choose meaningful names for variables.

Example:

```
# Good variable naming

user_name = "John"
total_items = 42


# Avoid using reserved words

class = "Python"  # Not recommended



# Use meaningful names

a = 10  # Less clear
num_of_students = 10    # More descriptive
```

Practice Exercises and Examples:

Example: Using Variables to Store and Manipulate Configuration Data in a DevOps Context

In a DevOps context, you often need to manage configuration data for various services or environments. Variables are essential for this purpose. Let's consider a scenario where we need to store and manipulate configuration data for a web server.

```
# Define configuration variables for a web server

server_name = "my_server"
port = 80
is_https_enabled = True
max_connections = 1000


# Print the configuration

print(f"Server Name: {server_name}")
print(f"Port: {port}")
print(f"HTTPS Enabled: {is_https_enabled}")
print(f"Max Connections: {max_connections}")


# Update configuration values

port = 443
is_https_enabled = False


# Print the updated configuration

print(f"Updated Port: {port}")
print(f"Updated HTTPS Enabled: {is_https_enabled}")
```

In this example, we use variables to store and manipulate configuration data for a web server. This allows us to easily update and manage the server's configuration in a DevOps context.

Python Functions, Modules and Packages

1. Differences Between Functions, Modules, and Packages

Functions

A function in Python is a block of code that performs a specific task. Functions are defined using the def keyword and can take inputs, called arguments. They are a way to encapsulate and reuse code.

Example:

```
def greet(name):
   return f"Hello, {name}!"

message = greet("Alice")
print(message)
```

In this example, *greet* is a function that takes a *name* argument and returns a greeting message.

Modules

A module is a Python script containing Python code. It can define functions, classes, and variables that can be used in other Python scripts. Modules help organize and modularize your code, making it more maintainable.

Example:

```
# my_module.py
def square(x):
    return x ** 2

pi = 3.14159265
```

You can use this module in another script:

```
import my_module

result = my_module.square(5)
print(result)
print(my_module.pi)
```

In this case, *my_module* is a Python module containing the *square* function and a variable *pi*.

Packages
A package is a collection of modules organized in directories. Packages help you organize related modules into a hierarchy. They contain a special file named *__init__.py*, which indicates that the directory should be treated as a package.

Example:
Suppose you have a package structure as follows:

```
my_package/
    __init__.py
    module1.py
    module2.py
```

You can use modules from this package as follows:

```
from my_package import module1
result = module1.function_from_module1()
```

In this example, *my_package* is a Python package containing modules *module1* and *module2.*

2. How to Import a Package :

Importing a package or module in Python is done using the *import* statement. You can import the entire package, specific modules, or individual functions/variables from a module.

Example:

```
# Import the entire module
import math

# Use functions/variables from the module
result = math.sqrt(16)
print(result)
```

```
# Import specific function/variable from a module
from math import pi
print(pi)
```

In this example, we import the *math* module and then use functions and variables from it. You can also import specific elements from modules using the *from module import element* syntax.

3. Python Workspaces

Python workspaces refer to the environment in which you develop and run your Python code. They include the Python interpreter, installed libraries, and the current working directory. Understanding workspaces is essential for managing dependencies and code organization.

Python workspaces can be local or virtual environments. A local environment is the system-wide Python installation, while a virtual environment is an isolated environment for a specific project. You can create virtual environments using tools like *virtualenv* or *venv.*

Example:

```
# Create a virtual environment
python -m venv myenv

# Activate the virtual environment (on Windows)
myenv\Scripts\activate

# Activate the virtual environment (on macOS/Linux)
source myenv/bin/activate
```

Once activated, you work in an isolated workspace with its Python interpreter and library dependencies.

Introduction to Operators in Python

Operators in Python are special symbols or keywords that are used to perform operations on variables and values. Python supports a wide range of operators, categorized into several types. These operators allow you to perform tasks such as arithmetic calculations, assign values to variables, compare values, perform logical operations, and more.
Here are the main types of operators in Python:

1. Arithmetic Operators: These operators are used for performing basic mathematical operations such as addition, subtraction, multiplication, division, and more.
2. Assignment Operators: Assignment operators are used to assign values to variables. They include the equal sign (=) and various compound assignment operators.
3. Relational Operators: Relational operators are used to compare values and determine the relationship between them. They return a Boolean result (True or False).

4. Logical Operators: Logical operators are used to combine and manipulate Boolean values. They include "and," "or," and "not."
5. Identity Operators: Identity operators are used to check if two variables point to the same object in memory. The identity operators in Python are "is" and "is not."
6. Membership Operators: Membership operators are used to check if a value is present in a sequence or collection, such as a list, tuple, or string. The membership operators in Python are "in" and "not in."
7. Bitwise Operators: Bitwise operators are used to perform operations on individual bits of binary numbers. They include bitwise AND, OR, XOR, and more.
8. Precedence of Operations: Operators in Python have different levels of precedence, which determine the order in which operations are performed in an expression.

Arithmetic Operations in Python

Introduction

Arithmetic operators in Python allow you to perform basic mathematical calculations on numeric values. These operators include addition, subtraction, multiplication, division, and more.

List of Arithmetic Operators

1. Addition (+): Adds two numbers.
2. Subtraction (-): Subtracts the right operand from the left operand.
3. Multiplication (*): Multiplies two numbers.
4. Division (/): Divides the left operand by the right operand (results in a floating-point number).
5. Floor Division (//): Divides the left operand by the right operand and rounds down to the nearest whole number.
6. Modulus (%): Returns the remainder of the division of the left operand by the right operand.
7. Exponentiation ():** Raises the left operand to the power of the right operand.

Examples

Addition

```
a = 5
b = 3
result = a + b
print(result)  # Output: 8
```

Subtraction

```
x = 10
y = 7
result = x - y
print(result)  # Output: 3
```

Assignment Operations in Python

Introduction
Assignment operators in Python are used to assign values to variables. They include the basic assignment operator (=) and various compound assignment operators that perform an operation on the variable while assigning a value.

List of Assignment Operators

1. **Basic Assignment (=):** Assigns a value to a variable.
2. **Addition Assignment (+=):** Adds the right operand to the left operand and assigns the result to the left operand.
3. **Subtraction Assignment (-=):** Subtracts the right operand from the left operand and assigns the result to the left operand.
4. **Multiplication Assignment (*=):** Multiplies the left operand by the right operand and assigns the result to the left operand.
5. **Division Assignment (/=):** Divides the left operand by the right operand and assigns the result to the left operand.
6. **Floor Division Assignment (//=):** Performs floor division on the left operand and assigns the result to the left operand.
7. **Modulus Assignment (%=):** Calculates the modulus of the left operand and assigns the result to the left operand.
8. **Exponentiation Assignment (**=):** Raises the left operand to the power of the right operand and assigns the result to the left operand.

Examples

Basic Assignment
```
x = 5
```
Addition Assignment
```
y = 10
y += 3  # Equivalent to y = y + 3
```

Bitwise Operations in Python

Introduction
Bitwise operators in Python are used to perform operations on individual bits of binary numbers. These operators include bitwise AND, OR, XOR, and more.

List of Bitwise Operators

1. **Bitwise AND (&):** Performs a bitwise AND operation on the binary representations of the operands.
2. **Bitwise OR (|):** Performs a bitwise OR operation.
3. **Bitwise XOR (^):** Performs a bitwise XOR operation.
4. **Bitwise NOT (~):** Flips the bits of the operand, changing 0 to 1 and 1 to 0.
5. **Left Shift (<<):** Shifts the bits to the left by a specified number of positions.

6. **Right Shift (>>):** Shifts the bits to the right.

Examples

Bitwise AND

```
a = 5  # Binary: 0101
b = 3  # Binary: 0011
result = a & b  # Result: 0001 (Decimal: 1)
```

Bitwise OR

```
x = 10  # Binary: 1010
y = 7   # Binary: 0111
result = x | y  # Result: 1111 (Decimal: 15)
```

Identity Operations in Python

Introduction

Identity operators in Python are used to compare the memory locations of two objects to determine if they are the same object or not. The two identity operators are "is" and "is not."

List of Identity Operators

1. **is:** Returns True if both operands refer to the same object.
2. **is not:** Returns True if both operands refer to different objects.

Examples

is Operator

```
x = [1, 2, 3]
y = x  # y now refers to the same object as x
result = x is y
# result will be True
```

is not Operator

```
a = "hello"
b = "world"
result = a is not b
# result will be True
```

Logical Operations in Python

Introduction

Logical operators in Python are used to manipulate and combine Boolean values. These operators allow you to perform logical operations such as AND, OR, and NOT.

List of Logical Operators

1. **AND (and):** Returns True if both operands are True.

2. **OR (or):** Returns True if at least one of the operands is True.
3. **NOT (not):** Returns the opposite Boolean value of the operand.

Examples
AND Operator
x = True
y = False
result = x and y
# result will be False

OR Operator
a = True
b = False
result = a or b
# result will be True

Membership Operations in Python

Introduction
Membership operators in Python are used to check whether a value is present in a sequence or collection, such as a list, tuple, or string. The membership operators are "in" and "not in."

List of Membership Operators

1. **in:** Returns True if the left operand is found in the sequence on the right.
2. **not in:** Returns True if the left operand is not found in the sequence on the right.

Examples
in Operator
fruits = ["apple", "banana", "cherry"]
result = "banana" in fruits
# result will be True

not in Operator
colors = ["red", "green", "blue"]
result = "yellow" not in colors
# result will be True

Precedence of Operations in Python

Introduction
Precedence of operations in Python defines the order in which different types of operators are evaluated in an expression. Operators with higher precedence are evaluated first.
Examples

Arithmetic Precedence

```
result = 5 + 3 * 2
# Multiplication has higher precedence, so result is 11, not 16
```

Relational Operations in Python

Introduction

Relational operators in Python are used to compare two values and determine the relationship between them. These operators return a Boolean result, which is either True or False.

List of Relational Operators

1. **Equal to (==):** Checks if two values are equal.
2. **Not equal to (!=):** Checks if two values are not equal.
3. **Greater than (>):** Checks if the left operand is greater than the right operand.
4. **Less than (<):** Checks if the left operand is less than the right operand.
5. **Greater than or equal to (>=):** Checks if the left operand is greater than or equal to the right operand.
6. **Less than or equal to (<=):** Checks if the left operand is less than or equal to the right operand.

Examples

Equal to

```
a = 5
b = 5
result = a == b
# result will be True
```

Not equal to

```
x = 10
y = 7
result = x != y
# result will be True
```

Conditional Statements in Python

Conditional statements are a fundamental part of programming that allow you to make decisions and execute different blocks of code based on certain conditions. In Python, you can use if, elif (short for "else if"), and else to create conditional statements.

if Statement

The if statement is used to execute a block of code if a specified condition is True. If the condition is False, the code block is skipped.

```
if condition:
   # Code to execute if the condition is True
```

- Example:

```
x = 10
if x > 5:
   print("x is greater than 5")
```

elif Statement

The elif statement allows you to check additional conditions if the previous if or elif conditions are False. You can have multiple elif statements after the initial if statement.

```
if condition1:
   # Code to execute if condition1 is True
elif condition2:
   # Code to execute if condition2 is True
elif condition3:
   # Code to execute if condition3 is True
# ...
else:
   # Code to execute if none of the conditions are True
```

- Example:

```
x = 10
if x > 15:
   print("x is greater than 15")
elif x > 5:
   print("x is greater than 5 but not greater than 15")
else:
   print("x is not greater than 5")
```

else Statement

The else statement is used to specify a block of code to execute when none of the previous conditions (in the if and elif statements) are True.

```
if condition:
   # Code to execute if the condition is True
else:
   # Code to execute if the condition is False
```

- Example:

```
x = 3
if x > 5:
   print("x is greater than 5")
else:
   print("x is not greater than 5")
```

Python Operators Assignment

In this assignment, you will explore various Python operators and their usage. Please complete the following tasks.

Task 1: Arithmetic Operators

1. Create two variables a and b with numeric values.
2. Calculate the sum, difference, product, and quotient of a and b.
3. Print the results.

```
a = 10
b = 5

sum_result = a + b
difference_result = a - b
product_result = a * b
quotient_result = a / b

print("Sum:", sum_result)
print("Difference:", difference_result)
print("Product:", product_result)
print("Quotient:", quotient_result)
```

Task 2: Comparison Operators

1. Compare the values of a and b using the following comparison operators: <, >, <=, >=, ==, and !=.
2. Print the results of each comparison.

```
a = 10
b = 5

less_than = a < b
greater_than = a > b
less_than_or_equal = a <= b
greater_than_or_equal = a >= b
equal = a == b
not_equal = a != b

print("a < b:", less_than)
print("a > b:", greater_than)
print("a <= b:", less_than_or_equal)
print("a >= b:", greater_than_or_equal)
print("a == b:", equal)
print("a != b:", not_equal)
```

Task 3: Logical Operators

1. Create two boolean variables, x and y.
2. Use logical operators (and, or, not) to perform various logical operations on x and y.
3. Print the results.

```
x = True
y = False
```

```
and_result = x and y
or_result = x or y
not_result_x = not x
not_result_y = not y

print("x and y:", and_result)
print("x or y:", or_result)
print("not x:", not_result_x)
print("not y:", not_result_y)
```

Task 4: Assignment Operators

1. Create a variable total and initialize it to 10.
2. Use assignment operators (+=, -=, *=, /=) to update the value of total.
3. Print the final value of total.

```
total = 10

total += 5
total -= 3
total *= 2
total /= 4

print("Final total:", total)
```

Task 5: Bitwise Operators (Optional)

1. If you are comfortable with bitwise operators, perform some bitwise operations on integer values and print the results. If not, you can skip this task.

Task 6: Identity and Membership Operators

1. Create a list my_list containing a few elements.
2. Use identity operators (is and is not) to check if two variables are the same object.
3. Use membership operators (in and not in) to check if an element is present in my_list.
4. Print the results.

```
my_list = [1, 2, 3, 4, 5]

# Identity operators
a = my_list
b = [1, 2, 3, 4, 5]

is_same_object = a is my_list
is_not_same_object = b is not my_list

# Membership operators
element_in_list = 3 in my_list
element_not_in_list = 6 not in my_list

print("a is my_list:", is_same_object)
print("b is not my_list:", is_not_same_object)
print("3 in my_list:", element_in_list)
print("6 not in my_list:", element_not_in_list)
```

Understanding Lists and List Data Structure

What is a List?
A list is a fundamental data structure in programming that allows you to store a collection of items. Lists are ordered and can contain elements of various data types, such as numbers, strings, and objects.

Creating Lists
You can create a list in various programming languages. In Python, for example, you create a list using square brackets:

```
my_list = [1, 2, 3, 'apple', 'banana']
```

List Indexing
List elements are indexed, starting from 0 for the first element. You can access elements by their index.

```
first_element = my_list[0]  # Access the first element (1)
```

List Length
You can find the length of a list using the len() function.

```
list_length = len(my_list)  # Length of the list (5)
```

List Manipulation and Common List Operations

Appending to a List
You can add elements to the end of a list using the append() method.

```
my_list.append(4)  # Adds 4 to the end of the list
```

Removing from a List
You can remove elements by their value using the remove() method.

```
my_list.remove('apple')  # Removes 'apple' from the list
```

Slicing a List
Slicing allows you to create a new list from a subset of the original list.

```
subset = my_list[1:4]  # Creates a new list with elements at index 1, 2, and 3
```

Concatenating Lists
You can combine two or more lists to create a new list.

```
new_list = my_list + [5, 6]  # Concatenates my_list with [5, 6]
```

Sorting a List
You can sort a list in ascending or descending order using the sort() method.

my_list.sort() # Sorts the list in ascending order

Checking for an Element
You can check if an element exists in a list using the in keyword.
is_present = 'banana' in my_list # Checks if 'banana' is in the list (True)

Understanding Tuples

What is a Tuple?
A tuple is a data structure similar to a list, but unlike lists, tuples are immutable, meaning their contents cannot be changed after creation. Tuples are typically used for grouping related data.

Creating Tuples
You can create a tuple in various programming languages. In Python, for example, you create a tuple using parentheses:
my_tuple = (1, 2, 'apple', 'banana')

Tuple Indexing
Tuple elements are indexed, starting from 0 for the first element. You can access elements by their index, just like lists.
first_element = my_tuple[0] # Access the first element (1)

Tuple Length
You can find the length of a tuple using the len() function.
tuple_length = len(my_tuple) # Length of the tuple (4)

Common Tuple Operations

Accessing Tuple Elements
Tuples are immutable, so you can only access their elements.
second_element = my_tuple[1] # Access the second element (2)

Tuple Packing and Unpacking
You can pack multiple values into a tuple and unpack them into separate variables.
coordinates = (3, 4)
x, y = coordinates # Unpack the tuple into x and y (x=3, y=4)

Concatenating Tuples
You can concatenate two or more tuples to create a new tuple.
new_tuple = my_tuple + (3.14, 'cherry') # Concatenates my_tuple with a new tuple

Checking for an Element
You can check if an element exists in a tuple using the in keyword.
is_present = 'apple' in my_tuple # Checks if 'apple' is in the tuple (True)

Using Tuples for Multiple Return Values

Tuples are often used to return multiple values from a function.

```
def get_coordinates():
    return (3, 4)

x, y = get_coordinates()  # Unpack the returned tuple (x=3, y=4)
```

Differences Between Tuples and Lists

Tuples and lists are both common data structures used in programming, but they have some fundamental differences that make them suitable for different purposes. Let's explore these differences:

1. Mutability
List: Lists are mutable, meaning their elements can be added, removed, or modified after creation. You can use methods like append(), remove(), and pop() to change the contents of a list.
Tuple: Tuples are immutable, and once created, their elements cannot be changed, added, or removed. You can't use methods to modify the tuple.

2. Syntax
List: Lists are created using square brackets [ ]. Elements are separated by commas.

```
my_list = [1, 2, 3, 'apple', 'banana']
```

Tuple: Tuples are created using parentheses ( ). Elements are also separated by commas.

```
my_tuple = (1, 2, 'apple', 'banana')
```

3. Performance
List: Lists may have slightly slower performance compared to tuples because they are mutable. Modifying a list requires memory reallocation, which can be slower for large lists.
Tuple: Tuples have better performance, especially for read-only operations, because of their immutability. They do not require memory reallocation.

4. Use Cases
List: Lists are used when you need a collection of elements that can change, such as a dynamic list of items or data that needs to be modified.
Tuple: Tuples are used when you need an ordered collection of elements that should not change, such as representing a point in 2D space (x, y), or when you want to ensure the integrity of the data.

5. Iteration
List: You can use a for loop or other iteration methods to iterate over the elements of a list.

```
for item in my_list:
    # Process each item
```

Tuple: You can iterate over the elements of a tuple in the same way as lists using a for loop.

```
for item in my_tuple:
    # Process each item
```

6. Memory Usage
List: Lists generally consume more memory than tuples because they need to store additional information to support their mutability.
Tuple: Tuples consume less memory because they are immutable, and the interpreter can optimize memory usage.

Frequently Asked Questions

Q1: What is a list in Python, and how is it used in DevOps?
Answer: A list in Python is a collection of ordered and mutable elements. In DevOps, lists are often used to manage and manipulate data, such as configurations, server names, and deployment targets. For example, you can use a list to store a list of servers that need to be provisioned or configured.

Q2: How do you create a list in Python, and can you provide an example related to DevOps?
Answer: In Python, you create a list using square brackets []. Here's an example related to DevOps:
servers = ['web-server-01', 'db-server-01', 'app-server-01']
This list can be used to represent a list of servers in a DevOps environment.

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?
Answer: The key difference is mutability; lists are mutable, while tuples are immutable. In DevOps, if you need a collection of items that won't change (e.g., server configurations, deployment steps), you would use a tuple. If the data can change (e.g., a list of active servers, configuration settings that may be updated), you would use a list.

Q4: How can you access elements in a list, and provide a DevOps-related example?
Answer: You can access elements in a list by using their index. In a DevOps context, if you have a list of server names and want to access the first server, you would do the following:
servers = ['web-server-01', 'db-server-01', 'app-server-01']
first_server = servers[0]

Q5: How do you add an element to the end of a list in Python? Provide a DevOps example.
Answer: You can add an element to the end of a list using the append() method. In DevOps, if you want to add a new server to a list of servers, you can do this:
servers = ['web-server-01', 'db-server-01']
servers.append('app-server-01')
Now, servers will contain 'app-server-01'.

Q6: How can you remove an element from a list in Python, and can you provide a DevOps use case?
Answer: You can remove an element from a list using the remove() method. In a DevOps use case, you might want to remove a server from a list of servers that are no longer needed:
servers = ['web-server-01', 'db-server-01', 'app-server-01']
servers.remove('db-server-01')

Loops in Python (for and while)

Introduction

Loops are a fundamental concept in programming, and they allow you to perform repetitive tasks efficiently. In Python, there are two primary types of loops: "for" and "while."

For Loop

The "for" loop is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a set of statements for each item in the sequence. The loop continues until all items in the sequence have been processed.

Syntax:

```
for variable in sequence:
   # Code to be executed for each item in the sequence
```

Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
   print(fruit)
```

Output:

```
apple
banana
cherry
```

In this example, the loop iterates over the "fruits" list, and in each iteration, the "fruit" variable takes on the value of the current item in the list.

While Loop

The "while" loop continues to execute a block of code as long as a specified condition is true. It's often used when you don't know in advance how many times the loop should run.

Syntax:

```
while condition:
   # Code to be executed as long as the condition is true
```

Example:

```
count = 0
while count < 5:
   print(count)
   count += 1
```

Output:

```
0
1
2
3
4
```

In this example, the "while" loop continues to execute as long as the "count" is less than 5. The "count" variable is incremented in each iteration.

Loop Control Statements (break and continue)

Introduction
Loop control statements are used to modify the behavior of loops, providing greater control and flexibility during iteration. In Python, two primary loop control statements are "break" and "continue."

break Statement
The "break" statement is used to exit the loop prematurely. It can be applied to both "for" and "while" loops, allowing you to terminate the loop when a particular condition is met.
Example:

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
   if number == 3:
      break
   print(number)
```

Output:

```
1
2
```

In this example, the loop stops when it encounters the number 3.

continue Statement
The "continue" statement is used to skip the current iteration of the loop and proceed to the next one. It can be used in both "for" and "while" loops, enabling you to bypass certain iterations based on a condition.
Example:

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
   if number == 3:
      continue
   print(number)
```

Output:

```
1
2
4
5
```

In this example, the loop skips the iteration where the number is 3 and continues with the next iteration.

Practice Exercise - Automating Log File Analysis

Introduction
In this practice exercise, we use a "for" loop to automate the analysis of a log file and identify lines containing the word "error." This demonstrates how loops can be used to process data and extract relevant information efficiently.
Example:
```
log_file = [
  "INFO: Operation successful",
  "ERROR: File not found",
  "DEBUG: Connection established",
  "ERROR: Database connection failed",
]

for line in log_file:
  if "ERROR" in line:
    print(line)
```
Output:
```
ERROR: File not found
ERROR: Database connection failed
```
In this exercise, the loop iterates through the "log_file" list and prints lines containing the word "ERROR."

For Loop DevOps use-cases

1. **Server Provisioning and Configuration:**

DevOps engineers use "for" loops when provisioning multiple servers or virtual machines with the same configuration. For example, when setting up monitoring agents on multiple servers:
```
servers=("server1" "server2" "server3")
for server in "${servers[@]}"; do
  configure_monitoring_agent "$server"
done
```

2. **Deploying Configurations to Multiple Environments:**

When deploying configurations to different environments (e.g., development, staging, production), DevOps engineers can use a "for" loop to apply the same configuration changes to each environment:
```
environments=("dev" "staging" "prod")
for env in "${environments[@]}"; do
  deploy_configuration "$env"
done
```

3. **Backup and Restore Operations:**

Automating backup and restore operations is a common use case. DevOps engineers can use "for" loops to create backups for multiple databases or services and later restore them as needed.

```
databases=("db1" "db2" "db3")
for db in "${databases[@]}"; do
   create_backup "$db"
done
```

4. **Log Rotation and Cleanup:**

DevOps engineers use "for" loops to manage log files, rotate logs, and clean up older log files to save disk space.

```
log_files=("app.log" "access.log" "error.log")
for log_file in "${log_files[@]}"; do
   rotate_and_cleanup_logs "$log_file"
done
```

5. **Monitoring and Reporting:**

In scenarios where you need to gather data or perform checks on multiple systems, a "for" loop is handy. For example, monitoring server resources across multiple machines:

```
servers=("server1" "server2" "server3")
for server in "${servers[@]}"; do
   check_resource_utilization "$server"
done
```

6. **Managing Cloud Resources:**

When working with cloud infrastructure, DevOps engineers can use "for" loops to manage resources like virtual machines, databases, and storage across different cloud providers.

```
instances=("instance1" "instance2" "instance3")
for instance in "${instances[@]}"; do
   resize_instance "$instance"
done
```

While Loop DevOps Usecases

DevOps engineers often use "while" loops in various real-time use cases to automate, monitor, and manage infrastructure and deployments. Here are some practical use cases from a DevOps engineer's perspective:

1. **Continuous Integration/Continuous Deployment (CI/CD) Pipeline:**

DevOps engineers often use "while" loops in CI/CD pipelines to monitor the deployment status of applications. They can create a "while" loop that periodically checks the status of a deployment or a rolling update until it completes successfully or fails. For example, waiting for a certain number of pods to be ready in a Kubernetes deployment:

```
while kubectl get deployment/myapp | grep -q 0/1; do
   echo "Waiting for myapp to be ready..."
   sleep 10
done
```

2. **Provisioning and Scaling Cloud Resources:**

When provisioning or scaling cloud resources, DevOps engineers may use "while" loops to wait for the resources to be fully provisioned and ready. For instance, waiting for an Amazon EC2 instance to become available:

```
while ! aws ec2 describe-instance-status --instance-ids i-1234567890abcdef0 | grep -q "running"; do
  echo "Waiting for the EC2 instance to be running..."
  sleep 10
done
```

3. **Log Analysis and Alerting:**

DevOps engineers can use "while" loops to continuously monitor logs for specific events or errors and trigger alerts when a certain condition is met. For example, tailing a log file and alerting when an error is detected:

```
while true; do
  if tail -n 1 /var/log/app.log | grep -q "ERROR"; then
    send_alert "Error detected in the log."
  fi
  sleep 5
done
```

4. **Database Replication and Data Synchronization:**

DevOps engineers use "while" loops to monitor database replication and ensure data consistency across multiple database instances. The loop can check for replication lag and trigger corrective actions when necessary.

```
while true; do
  replication_lag=$(mysql -e "SHOW SLAVE STATUS\G" | grep "Seconds_Behind_Master" | awk
'{print $2}')
  if [ "$replication_lag" -gt 60 ]; then
    trigger_data_sync
  fi
  sleep 60
done
```

5. **Service Health Monitoring and Auto-Recovery:**

DevOps engineers can use "while" loops to continuously check the health of services and automatically trigger recovery actions when services become unhealthy.

```
while true; do
  if ! check_service_health; then
    restart_service
  fi
  sleep 30
done
```

Lists Part-2

01-convert-string-to-list.py

```
folder_paths = input("Enter a list of folder paths separated by spaces: ").split()
```

02-main-construct.py

```
def main():
   folder_paths = input("Enter a list of folder paths separated by spaces: ").split()
   print(folder_paths)

   # Print elements in the list
   #for folder_path in folder_paths:
   #    print(folder_path)

if __name__ == "__main__":
   main()
```

03-list-files-in-folders.py

```
import os

def list_files_in_folder(folder_path):
   try:
      files = os.listdir(folder_path)
      return files, None
   except FileNotFoundError:
      return None, "Folder not found"
   except PermissionError:
      return None, "Permission denied"

def main():
   folder_paths = input("Enter a list of folder paths separated by spaces: ").split()

   for folder_path in folder_paths:
      files, error_message = list_files_in_folder(folder_path)
      if files:
         print(f"Files in {folder_path}:")
         for file in files:
            print(file)
      else:
         print(f"Error in {folder_path}: {error_message}")

if __name__ == "__main__":
   main()
```

Dictionaries

Overview:
A dictionary in Python is a data structure that allows you to store and retrieve values using keys. It is also known as a hashmap or associative array in other programming languages. Dictionaries are implemented as hash tables, providing fast access to values based on their keys.

Creating a Dictionary:
```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
```

Accessing Values:
```
print(my_dict['name'])  # Output: John
```

Modifying and Adding Elements:
```
my_dict['age'] = 26  # Modifying a value
my_dict['occupation'] = 'Engineer'  # Adding a new key-value pair
```

Removing Elements:
```
del my_dict['city']  # Removing a key-value pair
```

Checking Key Existence:
```
if 'age' in my_dict:
    print('Age is present in the dictionary')
```

Iterating Through Keys and Values:
```
for key, value in my_dict.items():
    print(key, value)
```

Sets and Set Operations

Overview:
A set in Python is an unordered collection of unique elements. It is useful for mathematical operations like union, intersection, and difference.

Creating a Set:
```
my_set = {1, 2, 3, 4, 5}
```

Adding and Removing Elements:
```
my_set.add(6)  # Adding an element
my_set.remove(3)  # Removing an element
```

Set Operations:
```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

union_set = set1.union(set2)  # Union of sets
intersection_set = set1.intersection(set2)  # Intersection of sets
difference_set = set1.difference(set2)  # Difference of sets
```

Subset and Superset:
```
is_subset = set1.issubset(set2)  # Checking if set1 is a subset of set2
is_superset = set1.issuperset(set2)  # Checking if set1 is a superset of set2
```

Practice Exercises and Examples

Example: Managing a Dictionary of Server Configurations and Optimizing Retrieval
Scenario:
Suppose you are managing server configurations using a dictionary.

```
server_config = {
  'server1': {'ip': '192.168.1.1', 'port': 8080, 'status': 'active'},
  'server2': {'ip': '192.168.1.2', 'port': 8000, 'status': 'inactive'},
  'server3': {'ip': '192.168.1.3', 'port': 9000, 'status': 'active'}
}
```

Function for Retrieval:

```
def get_server_status(server_name):
  return server_config.get(server_name, {}).get('status', 'Server not found')
```

Example Usage:

```
server_name = 'server2'
status = get_server_status(server_name)
print(f"{server_name} status: {status}")
```

Lists vs. Sets

Lists

- **Ordered Collection:**
  - Lists are ordered collections of elements. The order in which elements are added is preserved.
  - Elements can be accessed by their index.
- my_list = [1, 2, 3, 4, 5]

```
print(my_list[0])  # Output: 1
```

- **Mutable:**
  - Lists are mutable, meaning you can modify their elements after creation.

```
my_list[1] = 10
```

- **Allows Duplicate Elements:**
  - Lists can contain duplicate elements.

```
my_list = [1, 2, 2, 3, 4]
```

- **Use Cases:**
  - Use lists when you need an ordered collection with the ability to modify elements.

Sets

- **Unordered Collection:**
  - Sets are unordered collections of unique elements. The order in which elements are added is not preserved.
  - Elements cannot be accessed by their index.

```
my_set = {1, 2, 3, 4, 5}
```

- **Mutable:**
    - Sets are mutable, meaning you can add and remove elements after creation.

```
my_set.add(6)
```

- **No Duplicate Elements:**
    - Sets do not allow duplicate elements. If you try to add a duplicate, it won't raise an error, but the set won't change.

```
my_set = {1, 2, 2, 3, 4}  # Results in {1, 2, 3, 4}
```

- **Use Cases:**
    - Use sets when you need an unordered collection of unique elements, and you want to perform set operations like union, intersection, and difference.

Common Operations:

- **Adding Elements:**
    - Lists use append() or insert() methods.
    - Sets use add() method.
- **Removing Elements:**
    - Lists use remove(), pop(), or del statement.
    - Sets use remove() or discard() methods.
- **Checking Membership:**
    - Lists use the in operator.
    - Sets use the in operator as well, which is more efficient for sets.

```
# Lists
if 3 in my_list:
   print("3 is in the list")

# Sets
if 3 in my_set:
   print("3 is in the set")
```

Choosing Between Lists and Sets

- **Use Lists When:**
    - You need to maintain the order of elements.
    - Duplicate elements are allowed.
    - You need to access elements by index.

- **Use Sets When:**
    - Order doesn't matter.
    - You want to ensure unique elements.
    - You need to perform set operations like union, intersection, or difference.

demo-github-integration.py

```
# 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/kubernetes/kubernetes/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}")
```

Practice Exercises and Examples

Example: Managing a Dictionary of Server Configurations and Optimizing Retrieval
Scenario:
Suppose you are managing server configurations using a dictionary.

```
server_config = {
    'server1': {'ip': '192.168.1.1', 'port': 8080, 'status': 'active'},
    'server2': {'ip': '192.168.1.2', 'port': 8000, 'status': 'inactive'},
    'server3': {'ip': '192.168.1.3', 'port': 9000, 'status': 'active'}
}
```

Function for Retrieval:

```python
def get_server_status(server_name):
    return server_config.get(server_name, {}).get('status', 'Server not found')
```

Example Usage:

```python
server_name = 'server2'
status = get_server_status(server_name)
print(f"{server_name} status: {status}")
```

In this example, the function get_server_status optimizes the retrieval of the server status by using the get method and providing a default value if the server name is not found.

practicals.py

```python
# Server configurations dictionary
server_config = {
    'server1': {'ip': '192.168.1.1', 'port': 8080, 'status': 'active'},
    'server2': {'ip': '192.168.1.2', 'port': 8000, 'status': 'inactive'},
    'server3': {'ip': '192.168.1.3', 'port': 9000, 'status': 'active'}
}

# Retrieving information
def get_server_status(server_name):
    return server_config.get(server_name, {}).get('status', 'Server not found')

# Example usage
server_name = 'server2'
status = get_server_status(server_name)
print(f"{server_name} status: {status}")
```

server.conf

```
# Server Configuration File

# Network Settings
PORT = 8080
MAX_CONNECTIONS=600
TIMEOUT = 30

# Security Settings
SSL_ENABLED = true
SSL_CERT = /path/to/certificate.pem

# Logging Settings
LOG_LEVEL = INFO
LOG_FILE = /var/log/server.log

# Other Settings
ENABLE_FEATURE_X = true
```

update_server.py

```
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)
```

Github-JIRA intergration Project

create-jira.py

```
# This code sample uses the 'requests' library:

# http://docs.python-requests.org
import requests
from requests.auth import HTTPBasicAuth
import json

url = "https://veeramallaabhishek.atlassian.net/rest/api/3/issue"

API_TOKEN = ""

auth = HTTPBasicAuth("", 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": {
      "key": "AB"
    },
    "issuetype": {
      "id": "10006"
    },
    "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=(",", ": ")))
```

list_projects.py

```
# This code sample uses the 'requests' library:

# http://docs.python-requests.org
import requests
from requests.auth import HTTPBasicAuth
import json
```

```
url = "https://veeramallaabhishek.atlassian.net/rest/api/3/project"

API_TOKEN=""

auth = HTTPBasicAuth("", 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)
```

github-jira.py

```
# This code sample uses the 'requests' library:
# 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://veeramallaabhishek.atlassian.net/rest/api/3/issue"

    API_TOKEN=""

    auth = HTTPBasicAuth("", 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": {
           "key": "AB"
        },
        "issuetype": {
            "id": "10006"
        },
        "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)
```

hello-world.py

```
from flask import Flask

app = Flask(__name__)
```

```
@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run("0.0.0.0")
```

Interview Questions (Beginner and Intermediate)

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

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.

How can you secure your Python code and scripts?

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

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
```

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)
```

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
```

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()
```

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.")
```

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]
```

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
```

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
```

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)
```

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
```

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
```

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
```