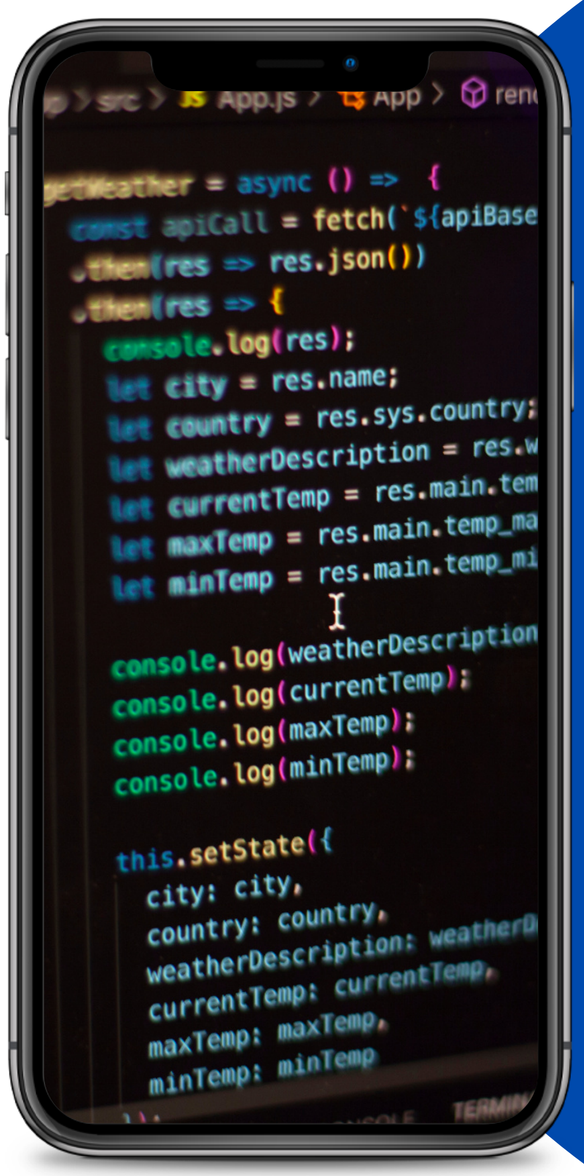
Network Journey

A journey towards packet life !!!

PYTHON NETWORK AUTOMATION for NETWORK ENGINEERS



INSTRUCTOR: MR. SAGAR
CCNP, DEVNET ASSOCIATE CERTIFIED

About Instructor

- Sagar holds CCNP and DEVNET Associate certifications & got industry-experience of upto 8 years working for top MNC and now working as full-time trainer for more then 5+ Years

- Sagar has trained more than 20,000+ students globally.

- Sagar is also Certified Udemy and Corporate trainer with 10+ years of Industrial experience.

Expertise in:
1 Enterprise
2. Security
3. Datacenter
4. Network Automation

Participant's Introduction

Break Time
10 mins

Network Journey
A journey towards packet life !!!

What are the Challenges in Daily Network Jobs?

Network Journey
A journey towards packet life !!!

Network Journey
A journey towards packet life !!!

What are the Challenges in Daily Network Jobs?

- COPYING FILES
- EXECUTING MULTIPLE COMMANDS
- UPGRADING NETWORK DEVICES
- REBOOTING DEVICES
- BACKUP / RESTORE
- CREATING REPORTS
- PUSHING CONFIGS
- PERFORMANCE & SLAs
- NETWORK ANALYSIS
- SCALING THE PROCESS
- DEALING WITH CLI COMMANDS
- HUMAN ERROR

SOLUTION IS

AUTOMATING BORING STUFF USING PYTHON

Why PYTHON ?

Network Journey
A journey towards packet life !!!

Worldwide, Apr 2023 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	27.43 %	-0.8 %
2		Java	16.41 %	-1.7 %
3		JavaScript	9.57 %	+0.3 %
4		C#	6.9 %	-0.3 %
5		C/C++	6.65 %	-0.5 %
6		PHP	5.17 %	-0.5 %
7		R	4.22 %	-0.4 %
8		TypeScript	2.89 %	+0.5 %
9	↑	Swift	2.31 %	+0.2 %
10	↓	Objective-C	2.09 %	-0.1 %

Apr 2023	Apr 2022	Change	Programming Language
1	1		Python
2	2		C
3	3		Java
4	4		C++

Top Computer Languages

As of 11 September 2021

PYPL Index: The PYPL "PopularitY of Programming Language Index" is created by analyzing how often language tutorials are searched on Google. The index is updated once a month.

TIOBE index: “The Importance Of Being Earnest”

PYTHON3

JAVA

C

Network Journey
A journey towards packet life !!!

Why Python?

- Easy to Learn and Use: Python has a simple and readable syntax, making it easier to understand and learn for beginners.
- Mature and Supportive Python Community: Python has a large and active community of developers, which means there are plenty of resources, libraries, and frameworks available to use and learn from.
- Support from Renowned Corporate Sponsors
- Hundreds of Python Libraries and Frameworks: Python has a vast collection of libraries and frameworks, such as NumPy, Pandas, Django, Flask, and TensorFlow, which make development faster and easier.
- Versatility, Efficiency, Reliability, and Speed
- Versatile: Python can be used for a wide range of applications, such as web development, data analysis, artificial intelligence, machine learning, and more.
- Cross-platform: Python code can be run on different operating systems like Windows, Linux, and macOS.
- Use of python in academics, DS, ML, Scientific Programming, Research, Big Data
- Open-source: Python is an open-source language, which means it is free to use and modify.

Network Journey
A journey towards packet life !!!

Interpreter vs Compiler?

An interpreter and a compiler are two different ways of executing code in a programming language. Here are the main differences:

1. Execution process: An interpreter reads the code line by line and executes it in real-time, whereas a compiler converts the entire code into machine code before execution.
2. Feedback: An interpreter provides immediate feedback as the code is executed, whereas a compiler provides feedback only after the entire code has been compiled.
3. Efficiency: Interpreted languages can be slower than compiled languages because the interpreter has to translate the code into machine code in real-time. Compiled languages, on the other hand, can be faster because the code is translated into machine code before execution.
4. Portability: Interpreted languages are generally more portable than compiled languages, as they can be run on multiple platforms without needing to be compiled for each platform.
5. Debugging: Interpreted languages are generally easier to debug because the interpreter can provide feedback in real-time as the code is executed. Compiled languages, on the other hand, can be more difficult to debug because the feedback is only provided after the code has been compiled.

A brief history of PYTHON 3

Network Journey
A journey towards packet life !!!

A brief history of PYTHON 3

Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to ABC programming language, which was inspired by SETL, capable of exception handling and interfacing with the Amoeba operating system.

Its implementation began in December 1989. Van Rossum shouldered sole responsibility for the project, as the lead developer, until 12 July 2018, when he announced his "permanent vacation" from his responsibilities as Python's Benevolent Dictator For Life, a title the Python community bestowed upon him to reflect his long-term commitment as the project's chief decision-maker.

In January 2019, active Python core developers elected a 5-member "Steering Council" to lead the project. As of 2021, the current members of this council are Barry Warsaw, Brett Cannon, Carol Willing, Thomas Wouters, and Pablo Galindo Salgado.

Apr 15, 2023 Python 3.11.3

2020 Python 2.0 EOL

2008 Python 3.0

2001 **UNICODE:** Python 2.0 introduced the Unicode string data type that allocated 16bit numbers to represent characters instead of standard 8bit strings. This adds 65,000 additional supported symbols from non-latin script languages like Russian, Chinese, or Arabic. It also added support for non-letter characters like emojis.

2000 Python 2 launched in 2000 with big changes to source code storage. It introduced many desired features like unicode support, list comprehension, and garbage collection. Major player: Java, C/C++

1991 Python 1 launched in 1994 with new features for functional programming, including lambda, map, filter and reduce. Major player: Perl

Network Journey
A journey towards packet life !!!

Python2 vs PYTHON 3

1. Print function: In Python 2, 'print' is a statement, while in Python 3, it is a function. This means that in Python 3, you need to use parentheses to enclose the arguments.

Python 2: print "Hello, World!"
Python 3: print("Hello, World!")

1. Integer division: In Python 2, dividing two integers results in an integer (floor division). In Python 3, dividing two integers results in a float.

Python 2: 5 / 2 returns 2
Python 3: 5 / 2 returns 2.5

1. Unicode support: Python 3 has better Unicode support, with strings being Unicode by default. In Python 2, strings are ASCII by default, and you need to use the 'u' prefix to create Unicode strings.

Python 2: **s = "Hello, World!"** (ASCII string), **u = u"Hello, World!"** (Unicode string)
Python 3: **s = "Hello, World!"** (Unicode string)

1. xrange function: In Python 2, there are two functions for creating ranges: 'range' and 'xrange'. 'range' returns a list, while 'xrange' returns an iterator. In Python 3, 'range' behaves like 'xrange', and 'xrange' has been removed.

1. Syntax changes: Python 3 introduced some syntax changes, such as 'nonlocal' for accessing non-local variables, the 'yield from' syntax for generators, and the 'async' and 'await' keywords for asynchronous programming.

1. Library changes: Many standard library modules and functions have been reorganized in Python 3 to make them more consistent and easier to use. Some functions have been removed or replaced, and others have been added.

1. End of life: Python 2 reached its end of life on January 1, 2020, which means that it no longer receives official support, updates, or bug fixes. Python 3 continues to be actively developed and maintained.

Network Journey
A journey towards packet life !!!

DIFFERENT WAYS TO AUTOMATE

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

FIRST WAY TO AUTOMATE

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

EEM (EMBEDDED EVENT MANAGER)

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

EXAMPLE:

```
event manager applet interface_Shutdown
event syslog pattern "Interface FastEthernet1/0,
changed state to administratively down"
action 1.0 cli command "enable"
action 1.5 cli command "config t"
action 2.0 cli command "interface E1/0"
action 2.5 cli command "no shutdown"
action 3.0 cli command "end"
action 3.5 cli command "who"
action 4.0 mail server "192.168.1.1" to
".engineer@cisco.com." from ".EEM@cisco.com."
subject ".ISP1_Interface_fa1/0_SHUT." body "Current
users $_cli_result"
```

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

SECOND WAY TO AUTOMATE

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

PYTHON, ANSIBLE, TERRAFORM, CHEF, PUPPET

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

EXAMPLE:

```
from netmiko import ConnectHandler
import getpass

cisco_device123 = {
    "device_type": "cisco_ios",
    "ip": "X.X.X.X",
    "username": input("enter your username:"),
    "password": getpass.getpass(),
    "port": 22,
}
connection123 = ConnectHandler(**cisco_device123)
output123 = connection123.send_command('show ip int br')
print(output123)
```

DOWNLOAD 20 FREE SCRITPS:

www.networkjourney.com/free-20-basic-python3-network-automation-scripts-for-practicing/

Network Journey
A journey towards packet life !!!

THIRD WAY TO AUTOMATE

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

API (APPLICATION PROGRAMMING INTERFACE)

NETWORK JOURNEY

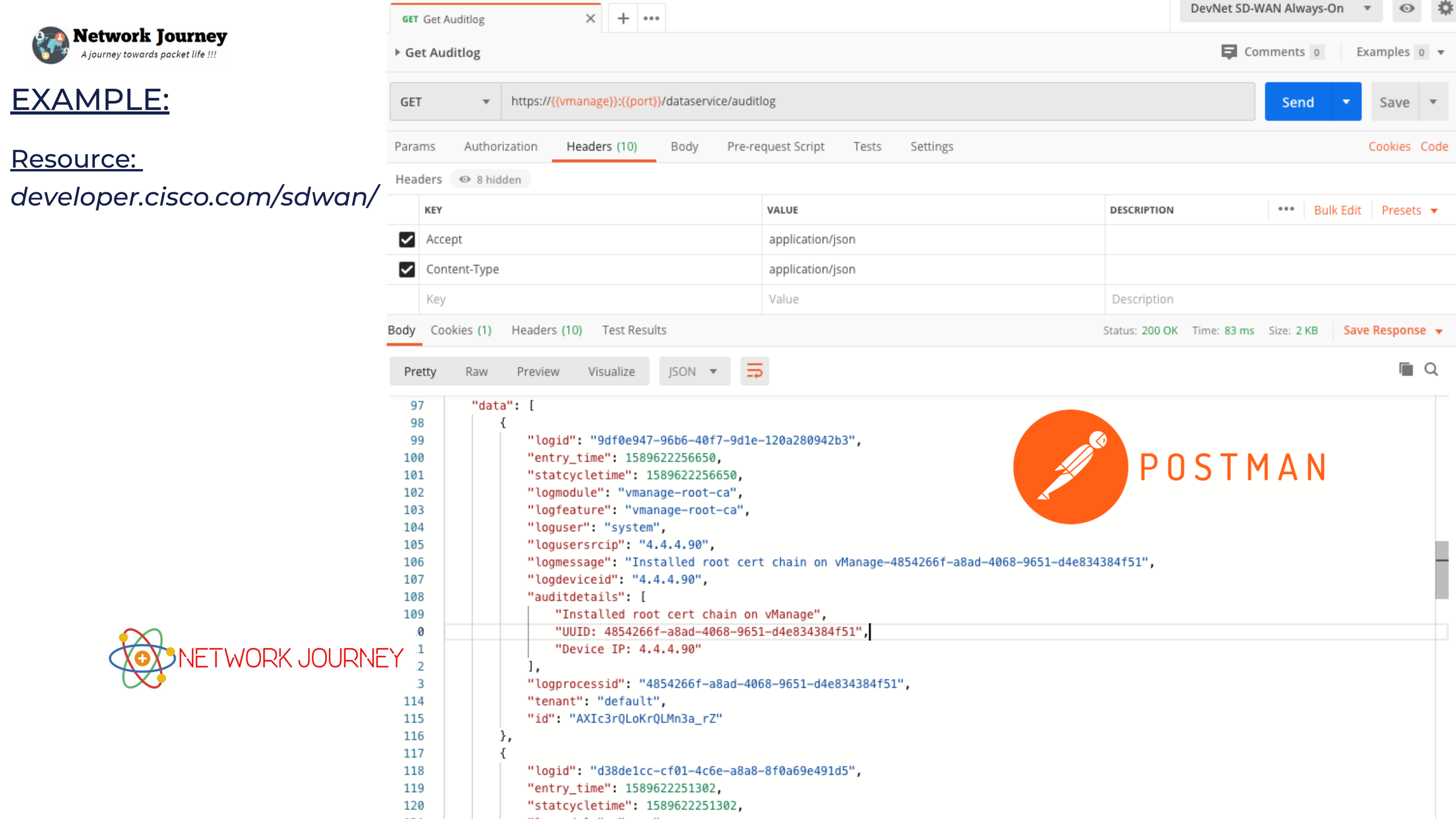
Network Journey
A journey towards packet life !!!

EXAMPLE:

Resource:

developer.cisco.com/sdwan/

NETWORK JOURNEY



APART FROM PYTHON SOME OTHER AVAILABLE TOOLS/LANGUAGE FOR NETWORK AUTOMATION

Network Journey
A journey towards packet life !!!

1. Ansible Benefits:
   - Agentless, works over SSH or other secure protocols
   - Simple, human-readable YAML-based language for scripting
   - Extensive library of pre-built modules for various tasks
   - Large community and wide support
   Disadvantages:
   - Performance can be slower compared to agent-based solutions
   - Limited support for Windows-based systems
2. Terraform Benefits:
   - Declarative language for defining infrastructure as code (IaC)
   - Supports a wide range of infrastructure providers, including cloud and on-premises platforms
   - Strong ecosystem and community
   Disadvantages:
   - Focused primarily on provisioning and infrastructure management, rather than configuration management
   - Steeper learning curve due to its domain-specific language (HCL)
3. Puppet Benefits:
   - Mature and widely used solution for configuration management
   - Declarative language for defining desired system state
   - Strong ecosystem and community
   - Supports multiple platforms, including Linux and Windows Disadvantages:
   - Requires a Puppet agent to be installed on each managed node
   - Complex, domain-specific language (DSL) can be challenging to learn
   - Network automation is less mature compared to other areas of Puppet
4. Chef Benefits:
   - Mature and widely used solution for configuration management
   - Ruby-based DSL for scripting
   - Strong ecosystem and community
   - Supports multiple platforms, including Linux and Windows Disadvantages:
   - Requires a Chef agent to be installed on each managed node
   - Steeper learning curve for non-Ruby users
   - Network automation is less mature compared to other areas of Chef

5. SaltStack (now Salt) Benefits:
   - High-performance and scalable due to its event-driven architecture
   - Python-based, allowing for powerful and flexible automation scripts
   - Strong ecosystem and community
   - Supports multiple platforms, including Linux and Windows Disadvantages:
   - Requires a Salt agent to be installed on each managed node
   - Steeper learning curve for non-Python users

BREAK TIME

- 15 min

Network Journey
A journey towards packet life !!!

PYTHON THEORY

Network Journey
A journey towards packet life !!!

Python print() Function

Example
Print a message onto the screen:
print("Hello World")

Definition and Usage
The print() function prints the specified message to the screen, or other standard output device.
The message can be a string, or any other object, the object will be converted into a string before written to the screen.

Syntax
print(object(s), sep=separator, end=end, file=file, flush=flush)

Parameter Values
- object(s) Any object, and as many as you like. Will be converted to string before printed
- sep='separator' Optional. Specify how to separate the objects, if there is more than one. Default is ' '
- end='end' Optional. Specify what to print at the end. Default is '/n' (line feed)
- file Optional. An object with a write method. Default is sys.stdout
- flush Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False

More Example
Print more than one object:
print("Hello", "how are you?")

Example
Print two messages, and specify the separator:
print("Hello", "how are you?", sep="---")

Example
x = ("ccna", "ccnp", "ccie")
print(x)

Network Journey
A journey towards packet life !!!

Python comments

Comments can be used to explain Python code.
Comments can be used to make the code more readable.
Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a #, and Python will ignore them:

Example

```
#This is a comment
print("Hello, World!")
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

Example

```
print("Hello, World!") #This is a comment
```

A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code:

Example

```
#print("Hello, World!")
print("Cheers, Mate!")
```

Multi Line Comments

Python does not really have a syntax for multi line comments.
To add a multiline comment you could insert a # for each line:

Example

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Or, not quite as intended, you can use a multiline string.
Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

Example

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

Network Journey
A journey towards packet life !!!

Python variable

Variables

Variables are containers for storing data values.

Creating Variables

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "Networkjourney"
print(x)
print(y)
```

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Multi Words Variable Names

Variable names with more than one word can be difficult to read.

There are several techniques you can use to make them more readable:

Camel Case

Each word, except the first, starts with a capital letter:

```
myVariableName = "Networkjourney"
```

Pascal Case

Each word starts with a capital letter:

```
MyVariableName = "Networkjourney"
```

Snake Case

Each word is separated by an underscore character:

```
my_variable_name = "Networkjourney"
```

Network Journey
A journey towards packet life !!!

Python Scope of Variable

In Python, variables are the containers for storing data values. Unlike other languages like C/C++/JAVA, Python is not “statically typed”. We do not need to declare variables before using them or declare their type. A variable is created the moment we first assign a value to it.

Python Scope variable

The location where we can find a variable and also access it if required is called the scope of a variable.

1. Python Local variable

Local variables are those that are initialized within a function and are unique to that function. It cannot be accessed outside of the function. Let’s look at how to make a local variable.

```
def test1():
    # local variable
    s = "I love Python"
    print(s)


# Driver code
test1()
print(s)
 --> this will fail if you try to print from outside of the function
```

Python Global variables

Global variables are the ones that are defined and declared outside any function and are not specified to any function. They can be used by any part of the program.

```
# This function uses global variable s
def f():
    print(s)


# Global scope
s = "I love Python"
f()
```

Global and Local Variables with the Same Name

Now suppose a variable with the same name is defined inside the scope of the function as well then it will print the value given inside the function only and not the global value.

```
# This function has a variable with
# name same as s.
def f():
    #s = "Content inside function."
    print(s)


# Global scope
s = "I love Python"
f()
print(s)
```

Network Journey
A journey towards packet life !!!

Python Scope of Variable

Python Scope variable

In Python, the nonlocal keyword is used in the case of nested functions. This keyword works similarly to the global, but rather than global, this keyword declares a variable to point to the variable of an outside enclosing function, in case of nested functions.

```
# nonlocal keyword

print("Value of a using nonlocal is : ", end="")

def outer():
    a = 5

    def inner():
        #nonlocal a
        a = 10

    inner()
    print(a)

outer()
```

Network Journey
A journey towards packet life !!!

Python open()

Open a file and print the content:

```
f = open("demofile.txt", "r")
print(f.read())
```

Definition and Usage

The open() function opens a file, and returns it as a file object.

A string, define which mode you want to open the file in:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exist

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Example1:

```
#OPEN()
#open is used with operations

xyz = open("config123455.txt", "w")
banner = """
THIS IS AUTHORIZED CISCO BOX
NOT TO BE USED BY UNAUTHORIZED USERS
"""
z = xyz.write(banner)
xyz.close()

xyz1 = open("config123455.txt", "r")
print(xyz1.read())

#r = read only ||| not going to create new file
```

Network Journey
A journey towards packet life !!!

Python data types

Built-in Data Types

In programming, data type is an important concept.
Variables can store data of different types, and different types can do different things.
Python has the following data types built-in by default, in these categories:

Text Type: str
Numeric Types: int, float, complex
Sequence Types: list, tuple, range
Mapping Type: dict
Set Types: set, frozenset
Boolean Type: bool
Binary Types: bytes, bytearray, memoryview

Getting the Data Type

You can get the data type of any object by using the type() function:
Example
Print the data type of the variable x:

```
x = 5
print(type(x))
```

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- List is a collection which is ordered and changeable. Allows duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered and unindexed. No duplicate members.
- Dictionary is a collection which is ordered* and changeable. No duplicate members.

Network Journey
A journey towards packet life !!!

Python strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.
'hello' is the same as "hello".
You can display a string literal with the print() function:

Example:
```
print("Hello")
print('Hello')
```

Assign String to a Variable
Assigning a string to a variable is done with the variable name followed by an equal sign and the string:
Example
```
a = "Hello"
print(a)
```

String Concatentation:
```
print("Hello/n" + "how are you?/n")
```

Multiline Strings
You can assign a multiline string to a variable by using three quotes:

Example
```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
ut labore et dolore magna aliqua."""
```

Network Journey
A journey towards packet life !!!

Python Built-in strings Operation

Here are some commonly used string operations and methods:

1. Concatenation: Combining two strings into one. Example: s1 = "Hello, " + "World!"
2. Repetition: Repeating a string multiple times. Example: s2 = "abc" * 3
3. Length: Finding the length of a string using the 'len()' function. Example: length = len("Hello, World!")
4. Indexing: Accessing individual characters in a string using indices. Example: first_char = "Hello, World!"[0]
5. Slicing: Extracting a substring from a string by specifying a start and end index. Example: substring = "Hello, World!"[0:5]
6. str.lower(): Convert all characters in a string to lowercase. Example: lowercase = "Hello, World!".lower()
7. str.upper(): Convert all characters in a string to uppercase. Example: uppercase = "Hello, World!".upper()
8. str.capitalize(): Capitalize the first character of a string and make the rest lowercase. Example: capitalized = "hello, world!".capitalize()
9. str.title(): Capitalize the first character of each word in a string. Example: title = "hello, world!".title()
10. str.strip(): Remove whitespace characters from the beginning and end of a string. Example: stripped = " Hello, World! ".strip()
11. str.lstrip(): Remove whitespace characters from the beginning of a string. Example: lstripped = " Hello, World! ".lstrip()
12. str.rstrip(): Remove whitespace characters from the end of a string. Example: rstripped = " Hello, World! ".rstrip()
13. str.replace(old, new): Replace all occurrences of a substring with another substring. Example: replaced = "Hello, World!".replace("World", "Python")
14. str.split(separator): Split a string into a list of substrings based on a separator. Example: words = "Hello, World!".split(" ")
15. str.join(iterable): Join a list of strings into a single string using a separator. Example: sentence = " ".join(["Hello", "World!"])
16. str.find(substring): Find the index of the first occurrence of a substring or return -1 if not found. Example: index = "Hello, World!".find("World")
17. str.startswith(prefix): Check if a string starts with a specified prefix. Example: starts_with_hello = "Hello, World!".startswith("Hello")
18. str.endswith(suffix): Check if a string ends with a specified suffix. Example: ends_with_world = "Hello, World!".endswith("World!")
19. str.count(substring): Count the number of non-overlapping occurrences of a substring in a string. Example: count_of_l = "Hello, World!".count("l")
20. str.format(): Format a string by replacing placeholders with specified values. Example: formatted = "Hello, {}!".format("Python")

These are some of the built-in string operations and methods in Python. There are many more methods available, and you can explore the official Python documentation to learn about them: https://docs.python.org/3/library/stdtypes.html#string-methods

Network Journey
A journey towards packet life !!!

Python Numbers

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

```
x = 1    # int
y = 2.8 # float
z = 1j  # complex
```

Network Journey
A journey towards packet life !!!

Python list

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- List is a collection which is ordered and changeable. Allows duplicate members.
- Dictionary is a collection which is ordered and changeable. No duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered and unindexed. No duplicate members.

Network Journey
A journey towards packet life !!!

Python list

LIST
ORDERED format (indexable)
CHANGEABLE ALLOWED
DUPLICATE ALLOWED

DUPLICATE ALLOWED

```
xyz123= ["1.1.1.1", "2.2.2.2", '3.3.3.3']
print(xyz123)
```

```
xyz123= ["1.1.1.1", "2.2.2.2", '3.3.3.3']
print(len(xyz123))
```

String and Number inside List array:

```
list1 = ["abc", 34, True, 40, "male"]
```

type()

```
xyz123 = ["1.1.1.1", "2.2.2.2", '3.3.3.3']
print(type(xyz123))
```

CHANGEABLE ALLOWED

```
xyz123 = ["1.1.1.1", "2.2.2.2", '3.3.3.3']
xyz123[1] = "192.168.1.1"
print(xyz123)
```

ORDERED format (indexable)

```
#list
xyz123 = ["1.1.1.1", "2.2.2.2", '3.3.3.3']
print(xyz123)
```

```
#in SET, ordered not seen
xyz123 = {"1.1.1.1", "2.2.2.2", '3.3.3.3'}
print(xyz123)
```

Python inbuilt Operations for list

list.append(x): Adds an element x to the end of the list.
pythonCopy code
numbers = [1, 2, 3]
numbers.append(4)
print(numbers) # Output: [1, 2, 3, 4]

list.extend(iterable): Appends the elements from the given iterable (e.g., another list, tuple, or string) to the end of the list.
pythonCopy code
numbers = [1, 2, 3]
numbers.extend([4, 5, 6])
print(numbers) # Output: [1, 2, 3, 4, 5, 6]

list.insert(i, x): Inserts an element x at the specified index i. If the index is greater than the list length, the element is appended to the end.
pythonCopy code
numbers = [1, 2, 4]
numbers.insert(2, 3)
print(numbers) # Output: [1, 2, 3, 4]

list.remove(x): Removes the first occurrence of the element x from the list. Raises a ValueError if the element is not found.
pythonCopy code
numbers = [1, 2, 3, 2]
numbers.remove(2)
print(numbers) # Output: [1, 3, 2]

list.pop([i]): Removes and returns the item at the specified index i. If no index is provided, it removes and returns the last item in the list.
pythonCopy code
numbers = [1, 2, 3]
last_item = numbers.pop()
print(last_item) # Output: 3print(numbers) # Output: [1, 2]

Network Journey
A journey towards packet life !!!

Python inbuilt Operations for list

1. list.clear(): Removes all items from the list.
2. pythonCopy code
3. numbers = [1, 2, 3]
4. numbers.clear()
5. print(numbers) # Output: []
6.
7. list.index(x[, start[, end]]): Returns the index of the first occurrence of the element x in the list, optionally within the slice [start:end]. Raises a ValueError if the element is not found.
8. pythonCopy code
9. numbers = [1, 2, 3, 2]
10. index = numbers.index(2)
11. print(index) # Output: 1
12.
13. list.count(x): Returns the number of occurrences of the element x in the list.
14. pythonCopy code
15. numbers = [1, 2, 3, 2]
16. count = numbers.count(2)
17. print(count) # Output: 2
18.
19. list.sort(key=None, reverse=False): Sorts the list in-place, using an optional key function and a reverse flag to determine the sort order.
20. pythonCopy code
21. numbers = [3, 1, 2]
22. numbers.sort()
23. print(numbers) # Output: [1, 2, 3]
24.
25. list.reverse(): Reverses the elements of the list in-place.
26. pythonCopy code
27. numbers = [1, 2, 3]
28. numbers.reverse()

print(numbers) # Output: [3, 2, 1]

Network Journey
A journey towards packet life !!!

Python inbuilt Operations for list

1. till here day1

Network Journey
A journey towards packet life !!!

Python dictionary

```
deviceinfo = {
  "ip": "1.1.1.1",
  "username": "admin",
  "password: "cisco"
}
```

Dictionary
Dictionaries are used to store data values in key:value pairs.
A dictionary is a collection which is ordered, changeable and does not allow duplicates.

DUPLICATE NOT ALLOWED
Example

```
deviceinfo = {
    "ip": "1.1.1.1",
    "username": "admin",
    "password": "cisco",
    "password": "cisco"
}
print(deviceinfo)
```

INDEX
Print the "username" value of the dictionary:

```
deviceinfo = {
    "ip": "1.1.1.1",
    "username": "admin",
    "password": "cisco"
}
print(deviceinfo["username"])
```

CHANGE
You can change the value of a specific item by referring to its key name, Example :
Change the "username" to "TOM":

```
deviceinfo = {
    "ip": "1.1.1.1",
    "username": "admin",
    "password": "cisco"
}
deviceinfo["username"] = "TOM"
print(deviceinfo)
```

As per PEP468 from python3.6 dict has ordered dictionary aspects introduced:
reference: https://docs.python.org/3.6/whatsnew/3.6.html#new-dict-implementation
https://softwaremaniacs.org/blog/2020/02/05/dicts-ordered/

2. Accessing a value by key:

```css
value = my_dict['key1']
```

3. Adding or updating a key-value pair:

```arduino
my_dict['key3'] = 'value3'
```

4. Checking if a key is in the dictionary:

```bash
if 'key1' in my_dict:
    print("Key found")
```

5. Removing a key-value pair using `del`:

```css
del my_dict['key1']
```

6. Getting the value for a key with a default value using `get()`:

```csharp
value = my_dict.get('key1', 'default_value')
```

7. Getting the length of a dictionary using `len()`:

```scss
length = len(my_dict)
```

8. Iterating over a dictionary:
   - Iterating over keys:

```scss
for key in my_dict.keys():
    print(key)
```

   - Iterating over values:

```scss
for value in my_dict.values():
    print(value)
```

   - Iterating over key-value pairs:

```scss
for key, value in my_dict.items():
    print(key, value)
```

9. Merging two dictionaries using `update()`:

```scss
my_dict.update(another_dict)
```

10. Clearing all key-value pairs from a dictionary using `clear()`:

```scss
my_dict.clear()
```

11. Creating a shallow copy of a dictionary using `copy()`:

```go
new_dict = my_dict.copy()
```

12. Creating a dictionary from a list of keys and a default value using `fromkeys()`:

```perl
keys = ['key1', 'key2', 'key3']
default_value = 'value'
new_dict = dict.fromkeys(keys, default_value)
```

Network Journey
A journey towards packet life !!!

Python tuples

Tuple
Tuples are used to store multiple items in a single variable.
Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
A tuple is a collection which is ordered and unchangeable.
Tuples are written with round brackets.

ORDERED (INDEXABLE)

```
thistuple = ("1.1.1.1", "2.2.2.2", "3.3.3.3")
print(thistuple[1])
```

NOT CHANGEABLE BUT WORKAROUND
Once a tuple is created, you cannot change its values. Tuples are unchangeable. But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

```
thistuple = ("1.1.1.1", "2.2.2.2", "3.3.3.3")
thislist = list(thistuple)
thislist[1] = "192.168.1.1"
thistuple = tuple(thislist)
print(thistuple)
```

DUPLICATE ALLOWED

```
thistuple = ("1.1.1.1", "2.2.2.2", "3.3.3.3", "2.2.2.2")
print(thistuple)
```

output>>

```
('1.1.1.1', '2.2.2.2', '3.3.3.3', '2.2.2.2')
```

Network Journey
A journey towards packet life !!!

Python set

Sets are used to store multiple items in a single variable.
Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
A set is a collection which is both unordered and unindexed.
Sets are written with curly brackets.

DUPLICATE NOT ALLOWED
Duplicate values will be ignored:

```
thisset123 = {"1.1.1.1", "2.2.2.2", "3.3.3.3", "2.2.2.2"}
print(thisset123)
```

ACCESSING SET
We cannot use index method to access data stored inside using set method as it doesn't support but we can use for loop to do this

```
thisset123 = {"1.1.1.1", "2.2.2.2", "3.3.3.3", "2.2.2.2"}
for x in thisset123:
  print(x)
```

Remove "banana" by using the remove() method:

```
thisset123 = {"1.1.1.1", "2.2.2.2", "3.3.3.3", "2.2.2.2"}
thisset123.remove("3.3.3.3")
print(thisset123)
```

Network Journey
A journey towards packet life !!!

Python if else

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

An "if statement" is written by using the if keyword.

Example1

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

Exampe2/elif - "if the previous conditions were not true, then try this condition".

```
a = 33
b = 33
if b > a:
 print("b is greater than a")
elif a == b:
 print("a and b are equal")
```

Example3/else: The else keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```

Example4: You can also have an else without the elif:

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
else:
     print("b is not greater than a")
```

Network Journey
A journey towards packet life !!!

Python for loop iterations

Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example: Print each fruit in a fruit list:

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

The break Statement : With the break statement we can stop the loop before it has looped through all the items:

Example
Exit the loop when x is "banana":

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

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next:

Example
Do not print banana:

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

The range() Function

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example
Using the range() function:

```
for x in range(6):
  print(x)
```

Else in For Loop

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Python for loop iterations (contd2)

Nested Loops

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

The pass Statement

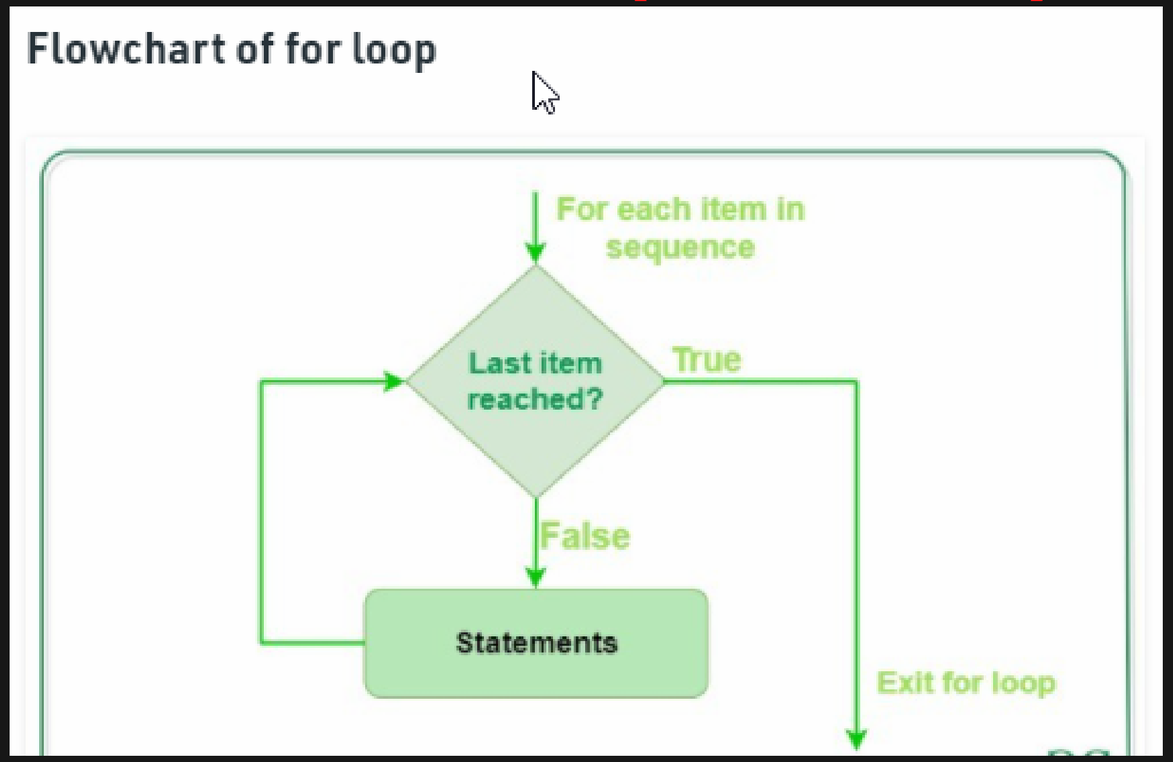
for loops cannot be empty, workaround is "pass"

Example

```
for x in [0, 1, 2]:
  pass
```

With Break statement

```
for char in 'networkjourney':
    # break the loop as soon it sees 'e'
    # or 't'
    if char == 't' or char == 'e':
        break

print('Current Character :', char)
```



p	q	p and q
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

p	q	p or q
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Network Journey
A journey towards packet life !!!

Python while loop iterations

The while Loop

With the while loop we can execute a set of statements as long as a condition is true

Example

Print i as long as i is less than 6:

```
i = 1
while i < 6:
  print(i)
  i += 1
```

The break Statement

With the break statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

The else Statement

With the else statement we can run a block of code once when the condition no longer is true:

Example

Print a message once the condition is false:

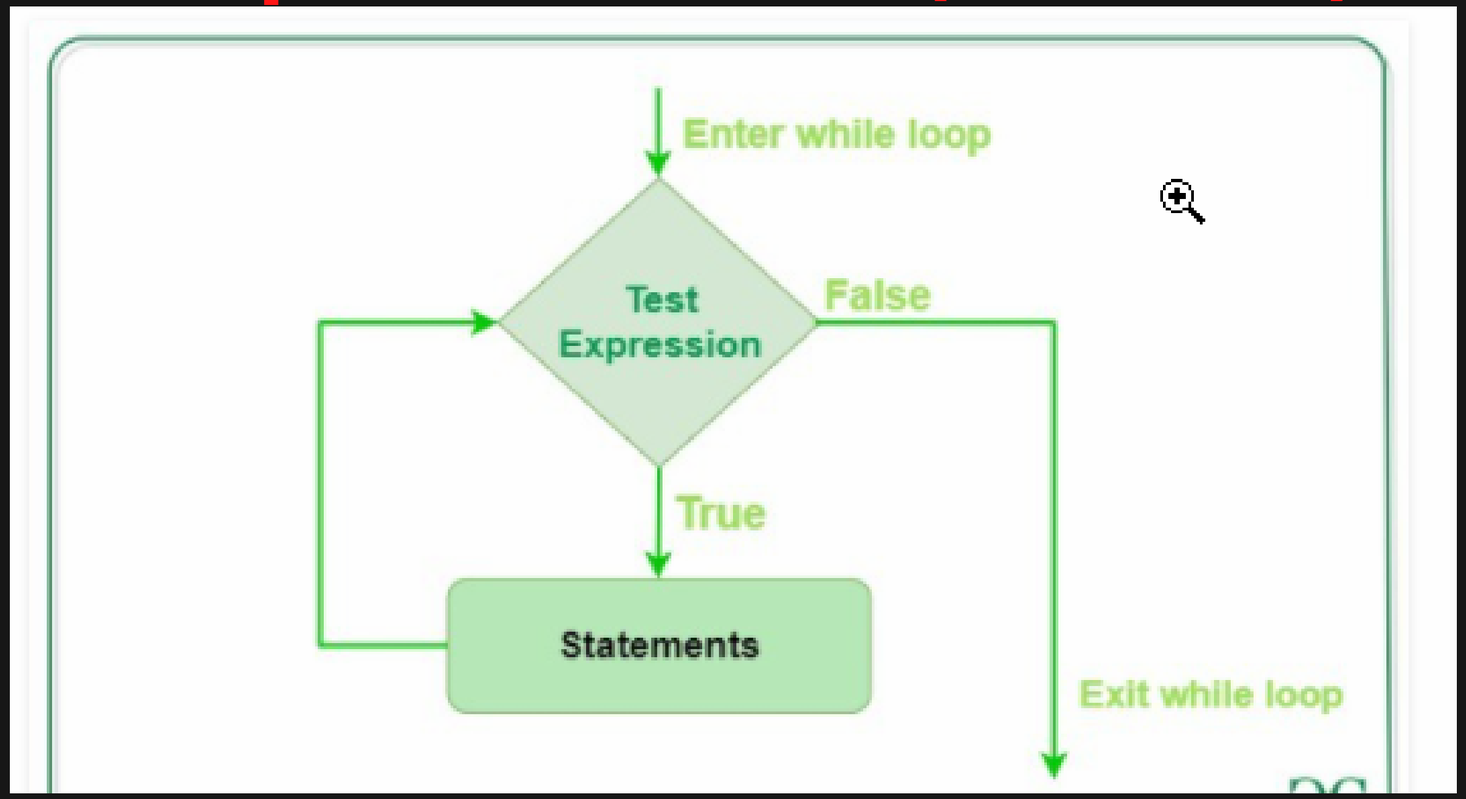
```
i = 1
while i < 6:
    print(i)
  i += 1
else:
    print("i is no longer less than 6")
```

Network Journey
A journey towards packet life !!!

Python while loop iterations (contd2)

INFINITE WHILE LOOP

```
var = 1
while var == 1:
   print("Hi!!!!")
```



Network Journey
A journey towards packet life !!!

Python Operators

Python Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values:

Example

```
print(10 + 5)
```

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

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

Network Journey
A journey towards packet life !!!

1. Arithmetic operators:
   - Addition (`+`): `3 + 4` => `7`
   - Subtraction (`-`): `9 - 2` => `7`
   - Multiplication (`*`): `3 * 5` => `15`
   - Division (`/`): `10 / 2` => `5.0`
   - Modulus (`%`): `10 % 3` => `1`
   - Floor division (`//`): `10 // 3` => `3`
   - Exponentiation (`**`): `2 ** 3` => `8`
2. Comparison operators:
   - Equal to (`==`): `3 == 3` => `True`
   - Not equal to (`!=`): `4 != 4` => `False`
   - Greater than (`>`): `5 > 3` => `True`
   - Less than (`<`): `2 < 1` => `False`
   - Greater than or equal to (`>=`): `7 >= 7` => `True`
   - Less than or equal to (`<=`): `3 <= 6` => `True`
3. Logical operators:
   - `and`: `True and False` => `False`
   - `or`: `True or False` => `True`
   - `not`: `not True` => `False`
4. Bitwise operators:
   - Bitwise AND (`&`): `5 & 3` => `1`
   - Bitwise OR (`|`): `5 | 3` => `7`
   - Bitwise XOR (`^`): `5 ^ 3` => `6`
   - Bitwise NOT (`~`): `~5` => `-6`
   - Left shift (`<<`): `5 << 1` => `10`
   - Right shift (`>>`): `5 >> 1` => `2`
5. Assignment operators:
   - `=`: `x = 5`
   - `+=`: `x += 3` => `x = x + 3`
   - `-=`: `x -= 2` => `x = x - 2`
   - `*=`: `x *= 4` => `x = x * 4`
   - `/=`: `x /= 2` => `x = x / 2`
   - `%=`: `x %= 3` => `x = x % 3`
   - `//=`: `x //= 3` => `x = x // 3`
   - `**=`: `x **= 2` => `x = x ** 2`
   - `&=`: `x &= 3` => `x = x & 3`
   - `|=`: `x |= 4` => `x = x | 4`
   - `^=`: `x ^= 5` => `x = x ^ 5`
   - `<<=`: `x <<= 2` => `x = x << 2`
   - `>>=`: `x >>= 2` => `x = x >> 2`
6. Membership operators:
   - `in`: `'a' in 'hello'` => `False`
   - `not in`: `'a' not in 'hello'` => `True`
7. Identity operators:
   - `is`: `x is y` => `True` if `x` and

1. Arithmetic operators:

- Addition (+):

```makefile
result = 3 + 4  # result is 7
```

- Subtraction (-):

```makefile
result = 10 - 3  # result is 7
```

- Multiplication (*):

```makefile
result = 4 * 3  # result is 12
```

- Division (/):

```makefile
result = 15 / 3  # result is 5.0
```

- Floor division (//):

```makefile
result = 17 // 3  # result is 5
```

- Modulus (%):

```makefile
result = 7 % 3  # result is 1
```

- Exponentiation (**):

```makefile
result = 2 ** 3  # result is 8
```

2. Comparison operators:

- Equal (==):

```sql
result = 5 == 5  # result is True
```

- Not equal (!=):

```sql
result = 5 != 6  # result is True
```

- Greater than (>):

```sql
result = 7 > 6  # result is True
```

- Less than (<):

```sql
result = 4 < 3  # result is False
```

- Greater than or equal to (>=):

```sql
result = 5 >= 5  # result is True
```

- Less than or equal to (<=):

```sql
result = 4 <= 3  # result is False
```

3. Logical operators:
   - AND (and):

```sql
result = (5 > 3) and (2 < 6)  # result is True
```

   - OR (or):

```sql
result = (5 < 3) or (2 < 6)  # result is True
```

   - NOT (not):

```sql
result = not (5 < 3)  # result is True
```

4. Bitwise operators:
   - AND (&):

```makefile
result = 5 & 3  # result is 1
```

   - OR (|):

```makefile
result = 5 | 3  # result is 7
```

   - XOR (^):

```makefile
result = 5 ^ 3  # result is 6
```

   - NOT (~):

```makefile
result = ~5  # result is -6
```

   - Left shift (<<):

```makefile
result = 5 << 2  # result is 20
```

   - Right shift (>>):

```makefile
result = 5 >> 2  # result is 1
```

Network Journey
A journey towards packet life !!!

5. Assignment operators:

- Assign (=):

```css
a = 5
```

- Add and assign (+=):

```css
a += 3  # equivalent to a = a + 3
```

- Subtract and assign (-=):

```css
a -= 2  # equivalent to a = a - 2
```

- Multiply and assign (*=):

```css
a *= 4  # equivalent to a = a * 4
```

- Divide and assign (/=):

```css
a /= 2  # equivalent to a = a / 2
```

Object Oriented Programming in Python

1. Class: A class is a blueprint for creating objects. It defines the attributes (data) and methods (functions) that objects of that class will have.

python Copy code

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("Woof!")
```

2. Object (Instance): An object or an instance is a specific occurrence of a class. It has its own set of attributes and can use the methods defined in the class.

python Copy code

```python
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)
```

3. Attributes: Attributes are the data or properties that belong to an object. In the `Dog` class example, `name` and `age` are attributes.

python Copy code

```
print(dog1.name)  # Output: Buddy
print(dog2.age)   # Output: 5
```

4. Methods: Methods are functions that belong to an object and can be called using the object. In the `Dog` class example, `bark` is a method.

python Copy code

```
dog1.bark()  # Output: Woof!
```

5. Inheritance: Inheritance is a way to create a new class that is a modified version of an existing class. The new class is called the subclass, and the existing class is the superclass. The subclass inherits attributes and methods from the superclass.

python Copy code

```
class ServiceDog(Dog):
    def __init__(self, name, age, service):
        super().__init__(name, age)
        self.service = service

    def perform_service(self):
        print(f"{self.name} is performing {self.service} service.")
```

FINAL OUTPUT

CLASS
OBJECT
ATTRIBUTES
METHODS
INHERITANCE

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("Woof!")

dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

print(dog1.name)  # Output: Buddy
print(dog2.name)

dog1.bark()  # Output: Woof!

class ServiceDog(Dog):
    def __init__(self, name, age, service):
        super().__init__(name, age)
        self.service = service

    def perform_service(self):
        print(f"{self.name} is performing {self.service} service.")

dog1 = ServiceDog("Buddy", 3, "bath")
dog2 = ServiceDog("Max", 5, "grooming")
print(dog2.service)

dog1.perform_service()
```

6. Encapsulation: Encapsulation refers to the idea of bundling data and methods that operate on that data within one unit (a class). Encapsulation is achieved by using private attributes and methods, which are not accessible from outside the class.

python Copy code

```python
class Example:
    def __init__(self, data):
        self.__data = data

    def get_data(self):
        return self.__data

    def set_data(self, value):
        self.__data = value
```

7. Polymorphism: Polymorphism allows us to use a single interface to represent different types of objects. It enables us to use a shared method name across different classes, each class implementing the method in a way that is specific to its own functionality.

python Copy code

```python
class Cat:
    def make_sound(self):
        print("Meow!")

class Duck:
    def make_sound(self):
        print("Quack!")

def make_animal_sound(animal):
    animal.make_sound()

cat1 = Cat()
duck1 = Duck()

make_animal_sound(cat1)  # Output: Meow!
make_animal_sound(duck1) # Output: Quack!
```

Exception & Error Handling

Types of Errors

There are three main types of errors in Python:

1. Syntax errors: These are errors that occur when the Python interpreter encounters a line of code that is not valid Python syntax. These errors are usually easy to identify, as they will produce a traceback that indicates the line number and type of error.
2. Runtime errors: These are errors that occur when the program is running. Also known as exceptions, they occur when the program encounters a situation that it cannot handle, such as dividing by zero or trying to access a variable that does not exist.
3. Semantic errors: These are errors that occur when the program is running, but the logic or behavior of the program is not what was intended. These errors can be difficult to identify and resolve, as they may not produce an error message or traceback.

Exception Handling

Python provides a built-in mechanism for handling exceptions, using the `try` and `except` statements. The basic syntax of a `try` and `except` block is as follows:

python Copy code

```python
try:
    # some code that may raise an exception
except ExceptionType:
    # code to handle the exception
```

The `try` block contains the code that may raise an exception, and the `except` block contains the code to handle the exception. The `ExceptionType` argument specifies the type of exception that the `except` block will handle. If an exception of the specified type is raised in the `try` block, the code in the `except` block will be executed.

Here's an example that demonstrates the use of a `try` and `except` block to handle a `ZeroDivisionError`:

python Copy code

```python
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

In this example, the `try` block attempts to divide 1 by 0, which raises a `ZeroDivisionError`. The `except` block catches the exception and prints a message indicating that division by zero is not allowed.

Handling Multiple Exceptions

You can handle multiple types of exceptions in a single `try` and `except` block by specifying multiple `ExceptionType` arguments, separated by commas:

python Copy code

```python
try:
    # some code that may raise an exception
except (ExceptionType1, ExceptionType2):
    # code to handle the exception
```

Alternatively, you can use multiple `except` blocks to handle different types of exceptions:

python Copy code

```python
try:
    # some code that may raise an exception
except ExceptionType1:
    # code to handle the first type of exception
except ExceptionType2:
    # code to handle the second type of exception
```

The `finally` Block

You can also include a **`finally`** block in a **`try`** and **`except`** statement. The **`finally`** block contains code that will always be executed, regardless of whether an exception was raised or not. This can be useful for cleaning up resources, such as closing files or network connections.

python Copy code

```python
try:
    # some code that may raise an exception
except ExceptionType:
    # code to handle the exception
finally:
    # code that will always be executed
```

Exception Handling using Debugging Mode (Pause on failure and resume post debug)

In Python, you can use the debugging mode to handle exceptions and errors during runtime. The debugging mode allows you to pause the program execution when an exception or error occurs, and then inspect the variables and the call stack to identify the cause of the problem.

To use the debugging mode in Python, you need to use a debugger. Python comes with a built-in debugger called `pdb`, which stands for Python Debugger. `pdb` allows you to step through the code, set breakpoints, and inspect the variables.

To enable the debugging mode in your Python script, you need to import the `pdb` module and call the `set_trace()` function at the point where you want to start the debugger.

python Copy code

```python
import pdb

def my_function():
    x = 1 / 0
    return x

pdb.set_trace()
result = my_function()
print(result)
```

In this example, the `pdb.set_trace()` call will start the debugger at the beginning of the `my_function()` function. When the program execution reaches this point, the debugger will pause the execution and show the debugging prompt:

scss Copy code

```
> <ipython-input-1-95e090c8b7ba>(5)my_function()
-> x = 1 / 0
(Pdb)
```

At this point, you can use various commands to inspect the variables and the call stack. For example, you can use the `p` command to print the value of a variable:

csharp Copy code

```
(Pdb) p x
*** ZeroDivisionError: division by zero
```

In this case, the `p x` command shows that a `ZeroDivisionError` has occurred.

You can also use the `c` command to continue the execution of the program until the next breakpoint or the end of the program:

```
r
```

Copy code

```
(Pdb) c
```

When the program execution continues, the `ZeroDivisionError` exception will be raised, but this time the debugger will catch it and show the traceback:

```
shell
```

Copy code

```
ZeroDivisionError: division by zero
> <ipython-input-1-95e090c8b7ba>(5)my_function()
-> x = 1 / 0
(Pdb)
```

At this point, you can use the `q` command to quit the debugger and terminate the program, or use the `c` command to continue the execution of the program until the end.

example#2

Here's an example that demonstrates how to use debugging mode to handle an exception:

python Copy code

```python
def divide(x, y):
    result = None
    try:
        result = x / y
    except ZeroDivisionError:
        import pdb; pdb.set_trace()
    return result

print(divide(1, 0))
```

In this example, the `divide` function attempts to divide `x` by `y`. If `y` is zero, a `ZeroDivisionError` is raised, and the program execution is paused using the `pdb.set_trace()` method.

When the program execution is paused, you can use the debugger to examine the state of the program. You can view the values of variables, inspect the call stack, and execute Python code to explore the program behavior.

To resume program execution after debugging, you can enter the `continue` command in the debugger. This will continue running the program until the next exception is encountered, or the program completes.

python Copy code

```
> /Users/user/test.py(6)divide()
-> result = x / y
(Pdb) y
0
(Pdb) continue
None
```

In this example, we can see that the value of `y` is `0`, which caused the `ZeroDivisionError`. To fix the problem, we could add a check to ensure that `y` is not zero before performing the division.

example#3

Here's an example of how to use Netmiko and the debugging mode to handle exceptions:

python Copy code

```python
import netmiko
import pdb

device = {
    'device_type': 'cisco_ios',
    'ip': '10.0.0.1',
    'username': 'admin',
    'password': 'password'
}

try:
    net_connect = netmiko.ConnectHandler(**device)
    pdb.set_trace()
    output = net_connect.send_command('show version')
    print(output)
except Exception as e:
    print(f"An error occurred: {e}")
finally:
    net_connect.disconnect()
```

When the program execution reaches the `pdb.set_trace()` call, the debugger will pause the execution and show the debugging prompt:

lua Copy code

```
> <ipython-input-1-95e090c8b7ba>(11)<module>()
-> output = net_connect.send_command('show version')
(Pdb)
```

At this point, we can use various commands to inspect the variables and the call stack, and identify the cause of any issues that may occur. For example, we can use the `p` command to print the value of a variable:

css Copy code

```
(Pdb) p output
```

We can also use the `c` command to continue the execution of the program until the next breakpoint or the end of the program:

r Copy code

```
(Pdb) c
```

example#4

python Copy code

```python
import pdb

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def divide(x, y):
    return x / y

pdb.set_trace()
try:
    a = 10
    b = 0
    c = add(a, b)
    d = subtract(a, b)
    e = divide(a, b)
except Exception as ex:
    print(f"An error occurred: {ex}")
```

In this example, we are defining three functions for addition, subtraction, and division, and we are using the `pdb.set_trace()` function to start the debugger just before the `try` block.

Inside the `try` block, we are calling the `add()`, `subtract()`, and `divide()` functions with some sample values. The `add()` and `subtract()` functions should execute without any issues, but the `divide()` function will raise a `ZeroDivisionError` exception when trying to divide by zero.

At this point, we can use the `n` command to execute the next line of code:

scss Copy code

```
(Pdb) n
> <ipython-input-1-95e090c8b7ba>(14)<module>()
-> b = 0
(Pdb)
```

We can then use the `n` command to execute the next line of code:

scss Copy code

```
(Pdb) n
> <ipython-input-1-95e090c8b7ba>(15)<module>()
-> c = add(a, b)
(Pdb)
```

At this point, we can use the `s` command to step into the `add()` function:

scss Copy code

```
(Pdb) s
--Call--
> <ipython-input-1-95e090c8b7ba>(3)add()
-> def add(x, y):
(Pdb)
```

Loggings and Examination of Debug Logs

Logging is an important concept in software development and is used to record information about the execution of a program. In Python, the `logging` module provides a powerful and flexible way to log messages from your application.

Here's an example of how to use the `logging` module to log messages and examine debug logs:

python Copy code

```python
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG,
                    format='%(asctime)s %(levelname)s %(message)s')

def add(x, y):
    logging.info(f"Adding {x} and {y}")
    return x + y

def subtract(x, y):
    logging.info(f"Subtracting {x} and {y}")
    return x - y

def divide(x, y):
    logging.info(f"Dividing {x} and {y}")
    try:
        return x / y
    except ZeroDivisionError as ex:
        logging.error(f"Error occurred: {ex}")
        return None

a = 10
b = 0

c = add(a, b)
d = subtract(a, b)
e = divide(a, b)

logging.debug(f"c={c}, d={d}, e={e}")
```

In this example, we are using the `basicConfig()` function to configure the logging module to write messages to a file named `example.log`. We are also setting the logging level to `DEBUG`, which means that all messages of level `DEBUG` and higher will be written to the log file.

Inside the functions, we are using the `logging.info()` and `logging.error()` functions to log messages at the `INFO` and `ERROR` levels, respectively. We are also using a `try` and

Data Types vs Data Structure

In Python, data types and data structures are related concepts, but they have distinct meanings:

1. Data types: Data types represent the type of data that can be stored and manipulated in a programming language. Python has several built-in data types, such as:

- int: Integer numbers (e.g., 1, 42, -7)
- float: Floating-point numbers (e.g., 3.14, -0.5, 1.0)
- str: Strings or sequences of characters (e.g., "hello", "Python")
- bool: Boolean values (True or False)
- NoneType: A special type representing the absence of a value (None)

These data types are the building blocks for creating more complex data structures.

1. Data structures: Data structures are ways of organizing and storing data so that they can be used effectively and efficiently. Python has several built-in data structures, which are collections of data types:

- Lists: Mutable, ordered collections of items (e.g., [1, 2, 3, "hello"])
- Tuples: Immutable, ordered collections of items (e.g., (1, 2, 3, "hello"))
- Sets: Unordered collections of unique items (e.g., {1, 2, 3, 4, 5})
- Dictionaries: Collections of key-value pairs, also known as associative arrays or hash tables (e.g., {"apple": 1, "banana": 2, "orange": 3})

Data structures allow you to manage and manipulate data more effectively, depending on the specific use case and the operations you need to perform.
In summary, data types represent the kind of data that can be stored and manipulated, while data structures are used to organize, store, and manage collections of data types in a way that suits particular programming needs.

Data Structure Overview

Data structures are fundamental concepts of computer science which helps is writing efficient programs in any language. Python is a high-level, interpreted, interactive and object-oriented scripting language using which we can study the fundamentals of data structure in a simpler way as compared to other programming languages.
In this chapter we are going to study a short overview of some frequently used data structures in general and how they are related to some specific python data types. There are also some data structures specific to python which is listed as another category.

General Data Structures

The various data structures in computer science are divided broadly into two categories shown below. We will discuss about each of the below data structures in detail in subsequent chapters.

Liner Data Structures

These are the data structures which store the data elements in a sequential manner.

- Array − It is a sequential arrangement of data elements paired with the index of the data element.
- Linked List − Each data element contains a link to another element along with the data present in it.
- Stack − It is a data structure which follows only to specific order of operation. LIFO(last in First Out) or FILO(First in Last Out).
- Queue − It is similar to Stack but the order of operation is only FIFO(First In First Out).
- Matrix − It is two dimensional data structure in which the data element is referred by a pair of indices.

Non-Liner Data Structures

These are the data structures in which there is no sequential linking of data elements. Any pair or group of data elements can be linked to each other and can be accessed without a strict sequence.

- Binary Tree − It is a data structure where each data element can be connected to maximum two other data elements and it starts with a root node.
- Heap − It is a special case of Tree data structure where the data in the parent node is either strictly greater than/ equal to the child nodes or strictly less than it’s child nodes.
- Hash Table − It is a data structure which is made of arrays associated with each other using a hash function. It retrieves values using keys rather than index from a data element.
- Graph − It is an arrangement of vertices and nodes where some of the nodes are connected to each other through links.

Python Specific Data Structures

These data structures are specific to python language and they give greater flexibility in storing different types of data and faster processing in python environment.

- List − It is similar to array with the exception that the data elements can be of different data types. You can have both numeric and string data in a python list.
- Tuple − Tuples are similar to lists but they are immutable which means the values in a tuple cannot be modified they can only be read.
- Dictionary − The dictionary contains Key-value pairs as its data elements.

Here are some examples of how to use Python's built-in data structures:

1. Lists:

Create a list:

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

Add an element to the end of the list:

```python
my_list.append(6)
```

Remove an element from the list:

```python
my_list.remove(3)
```

Access an element by index:

```python
print(my_list[2])  # Output: 4
```

2. Tuples:

Create a tuple:

```python
my_tuple = (1, 2, 3, 4, 5)
```

Access an element by index:

```python
print(my_tuple[1])  # Output: 2
```

3. Sets:

Create a set:

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

Add an element to the set:

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

Remove an element from the set:

```python
my_set.remove(3)
```

Find the union of two sets:

```python
other_set = {4, 5, 6, 7, 8}
union_set = my_set.union(other_set)
```

4. Dictionaries:

Create a dictionary:

```python
my_dict = {'apple': 1, 'banana': 2, 'orange': 3}
```

Add or update a key-value pair in the dictionary:

```python
my_dict['grape'] = 4
```

Remove a key-value pair from the dictionary:

```python
del my_dict['apple']
```

Access a value by its key:

```python
print(my_dict['banana'])  # Output: 2
```

Python Algorithms

Python's built-in functions and modules provide several algorithms that can be used to solve various problems. Here are some examples of using these algorithms in Python:

1. Sorting:

Sort a list in ascending order:

```python
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5]
my_list.sort()
print(my_list)  # Output: [1, 1, 2, 3, 4, 5, 5, 6, 9]
```

Sort a list in descending order using the `sorted` function:

```python
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5]
sorted_list = sorted(my_list, reverse=True)
print(sorted_list)  # Output: [9, 6, 5, 5, 4, 3, 2, 1, 1]
```

2. Binary search (using the `bisect` module):

```python
import bisect

my_list = [1, 3, 4, 4, 6, 8]

# Find the index where the element should be inserted to maintain the sorted
index = bisect.bisect_left(my_list, 4)
print(index)  # Output: 2
```

3. Finding the greatest common divisor (using the `math` module):

```python
import math

a = 56
b = 98
gcd = math.gcd(a, b)
print(gcd)  # Output: 14
```

4. Permutations and combinations (using the `itertools` module):

```python
import itertools

items = ['A', 'B', 'C']

# Find all possible permutations of the items
permutations = list(itertools.permutations(items))
print(permutations)  # Output: [('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A',

# Find all possible combinations of the items, taken 2 at a time
combinations = list(itertools.combinations(items, 2))
print(combinations)  # Output: [('A', 'B'), ('A', 'C'), ('B', 'C')]
```

Fundamentals of SDN (Software Defined Networking)

Software Defined Networking (SDN) is an approach to networking that aims to make networks more flexible, programmable, and centrally managed. SDN separates the network's control plane (which makes decisions about how traffic should flow) from the data plane (which forwards traffic according to those decisions). This separation allows for more dynamic control and management of network resources.

Here are some of the key fundamentals of SDN:

1. Centralized Control: In SDN, the control plane is managed by a central controller, which has a global view of the entire network. This centralized control allows for more efficient and intelligent management of network resources, as well as faster response to changes in network conditions.
2. Network Programmability: SDN enables network administrators to program the behavior of the network through software, rather than relying on manual configuration of individual devices. This programmability allows for more flexible and dynamic management of network resources, and it enables administrators to adapt the network to the specific needs of their applications and services.
3. Abstraction: SDN abstracts the underlying network infrastructure from the applications and services that use it, allowing them to interact with the network in a more simplified and consistent way. This abstraction enables application developers and network administrators to focus on the requirements of their applications, without having to worry about the details of the underlying network hardware.
4. Open Standards: SDN often relies on open standards and protocols, such as OpenFlow, to facilitate communication between the controller and network devices. This openness encourages interoperability between different vendors' hardware and software, and it fosters innovation by allowing developers to build upon existing standards and protocols.
5. Virtualization: SDN can be combined with network virtualization technologies, such as network function virtualization (NFV), to create virtual networks that can be easily created, modified, and deleted through software. These virtual networks can be customized to the specific needs of different applications and services, and they can be isolated from one another to improve security and performance.

Basics of Network Automation

Network Journey
A journey towards packet life !!!

What are the Challenges in Daily Network Jobs?

- COPYING FILES
- EXECUTING MULTIPLE COMMANDS
- UPGRADING NETWORK DEVICES
- REBOOTING DEVICES
- BACKUP / RESTORE
- CREATING REPORTS
- PUSHING CONFIGS
- PERFORMANCE & SLAs
- NETWORK ANALYSIS
- SCALING THE PROCESS
- DEALING WITH CLI COMMANDS
- HUMAN ERROR

Network Journey
A journey towards packet life !!!

DIFFERENT WAYS TO AUTOMATE

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

FIRST WAY TO AUTOMATE

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

EEM (EMBEDDED EVENT MANAGER)

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

EXAMPLE:

```
event manager applet interface_Shutdown
event syslog pattern "Interface FastEthernet1/0,
changed state to administratively down"
action 1.0 cli command "enable"
action 1.5 cli command "config t"
action 2.0 cli command "interface E1/0"
action 2.5 cli command "no shutdown"
action 3.0 cli command "end"
action 3.5 cli command "who"
action 4.0 mail server "192.168.1.1" to
".engineer@cisco.com." from ".EEM@cisco.com."
subject ".ISP1_Interface_fa1/0_SHUT." body "Current
users $_cli_result"
```

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

SECOND WAY TO AUTOMATE

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

PYTHON, ANSIBLE, TERRAFORM, CHEF, PUPPET

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

EXAMPLE:

```
from netmiko import ConnectHandler
import getpass

cisco_device123 = {
    "device_type": "cisco_ios",
    "ip": "X.X.X.X",
    "username": input("enter your username:"),
    "password": getpass.getpass(),
    "port": 22,
}
connection123 = ConnectHandler(**cisco_device123)
output123 = connection123.send_command('show ip int br')
print(output123)
```

DOWNLOAD 20 FREE SCRITPS:

www.networkjourney.com/free-20-basic-python3-network-automation-scripts-for-practicing/

Network Journey
A journey towards packet life !!!

Network Journey
A journey towards packet life !!!

THIRD WAY TO AUTOMATE

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

API (APPLICATION PROGRAMMING INTERFACE)

Network Journey
A journey towards packet life !!!

NETWORK JOURNEY

Network Journey
A journey towards packet life !!!

EXAMPLE:

Resource:

developer.cisco.com/sdwan/

GET Get Auditlog

DevNet SD-WAN Always-On

Get Auditlog

Comments 0 Examples 0

GET https://{{vmanage}}:{{port}}/dataservice/auditlog

Network Journey
A journey towards packet life !!!

Save

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies Code

Headers 8 hidden

KEY	VALUE	DESCRIPTION
Accept	application/json	
Content-Type	application/json	
Key	Value	Description

Body Cookies (1) Headers (10) Test Results Status: 200 OK Time: 83 ms Size: 2 KB Save Response

Pretty Raw Preview Visualize JSON


```
"data": [
    {
        "logid": "9df0e947-96b6-40f7-9d1e-120a280942b3",
        "entry_time": 1589622256650,
        "statcycletime": 1589622256650,
        "logmodule": "vmanage-root-ca",
        "logfeature": "vmanage-root-ca",
        "loguser": "system",
        "logusersrcip": "4.4.4.90",
        "logmessage": "Installed root cert chain on vManage-4854266f-a8ad-4068-9651-d4e834384f51",
        "logdeviceid": "4.4.4.90",
        "auditdetails": [
            "Installed root cert chain on vManage",
            "UUID: 4854266f-a8ad-4068-9651-d4e834384f51",
            "Device IP: 4.4.4.90"
        ],
        "logprocessid": "4854266f-a8ad-4068-9651-d4e834384f51",
        "tenant": "default",
        "id": "AXIc3rQLoKrQLMn3a_rZ"
    },
    {
        "logid": "d38de1cc-cf01-4c6e-a8a8-8f0a69e491d5",
        "entry_time": 1589622251302,
        "statcycletime": 1589622251302,
```

Network Journey
A journey towards packet life !!!

Implementing IOS Automation using Netmiko

Management Interface Configuration (cisco ios)

Network Journey
A journey towards packet life !!!

```
hostname R1
!
int fa0/0
no shutdown
ip add dhcp
exit
!
username admin pri 15 pass cisco
!
ip domain-name networkjourney.com
crypto key generate rsa modul 1024
!
line vty 0 4
login local
exec-timeout 0 5
transport input all
!
do wr mem
!
end
!
```

Netmiko is a popular Python library that simplifies the process of connecting to and automating network devices, such as routers and switches, including those running Cisco IOS. It is built on top of the Paramiko SSH library, and provides a higher-level API for interacting with network devices.

To get started with implementing IOS automation using Netmiko, follow these steps:

1. Install Netmiko:

You can install Netmiko using pip:

```
Copy code
pip install netmiko
```

2. Import the required libraries:

In your Python script, import the necessary libraries:

```python
Copy code
from netmiko import ConnectHandler
```

3. Define the device's connection details:

Create a dictionary containing the connection details for the device you want to connect to, such as its IP address, SSH username and password, and device type:

```python
Copy code
cisco_device = {
    'device_type': 'cisco_ios',
    'ip': '192.168.1.1',
    'username': 'your-username',
    'password': 'your-password',
}
```

Replace the 'ip', 'username', and 'password' values with the correct information for your device.

4. Establish a connection to the device:

Network Journey
A journey towards packet life !!!

Use the `ConnectHandler` function to establish an SSH connection to the device:

```python
Copy code
connection = ConnectHandler(**cisco_device)
```

5. Send commands to the device:

You can now use the `send_command` method to send commands to the device and retrieve the output:

```python
Copy code
output = connection.send_command('show interfaces')
print(output)
```

6. Send configuration commands:

To send configuration commands, you can use the `send_config_set` method. This method automatically enters and exits the configuration mode on the device:

```python
Copy code
config_commands = [
    'interface GigabitEthernet0/1',
    'description Uplink',
    'ip address 192.168.2.1 255.255.255.0',
]

output = connection.send_config_set(config_commands)
print(output)
```

Netmiko is a popular Python library that simplifies the process of connecting to and automating network devices, such as routers and switches, including those running Cisco IOS. It is built on top of the Paramiko SSH library, and provides a higher-level API for interacting with network devices.

To get started with implementing IOS automation using Netmiko, follow these steps:

1. Install Netmiko:

You can install Netmiko using pip:

```
pip install netmiko
```

2. Import the required libraries:

In your Python script, import the necessary libraries:

```python
from netmiko import ConnectHandler
```

3. Define the device's connection details:

Create a dictionary containing the connection details for the device you want to connect to, such as its IP address, SSH username and password, and device type:

```python
cisco_device = {
    'device_type': 'cisco_ios',
    'ip': '192.168.1.1',
    'username': 'your-username',
    'password': 'your-password',
}
```

Replace the 'ip', 'username', and 'password' values with the correct information for your device.

4. Establish a connection to the device:

Use the `ConnectHandler` function to establish an SSH connection to the device:

```python
connection = ConnectHandler(**cisco_device)
```

5. Send commands to the device:

You can now use the `send_command` method to send commands to the device and retrieve the output:

```python
output = connection.send_command('show interfaces')
print(output)
```

6. Send configuration commands:

To send configuration commands, you can use the `send_config_set` method. This method automatically enters and exits the configuration mode on the device:

```python
config_commands = [
    'interface GigabitEthernet0/1',
    'description Uplink',
    'ip address 192.168.2.1 255.255.255.0',
]

output = connection.send_config_set(config_commands)
print(output)
```

Network Journey
A journey towards packet life !!!

7. Disconnect from the device:

Once you're done, it's a good practice to close the connection:

```python
connection.disconnect()
```

Network Journey
A journey towards packet life !!!

Script#1


```
from netmiko import ConnectHandler
deviceinfo123 = {
    "device_type": "cisco_ios",
    "ip": "sandbox-iosxr-1.cisco.com",
    "username": "admin",
    "password": "C1sco12345"
}
ssh123 = ConnectHandler(**deviceinfo123)
#
output123 = ssh123.send_command("show ip int br")
print(output123)
```

```
from netmiko import ConnectHandler
deviceinfo123 = {
    "device_type": "cisco_ios",
    "ip": "sandbox-iosxr-1.cisco.com",
    "username": "admin",
    "password": "C1sco12345"
}
ssh123 = ConnectHandler(**deviceinfo123)
mycommands = ["show run | i hostname", "show ip int br", "show clock",
"show ver"]
for singleclicommands in mycommands:
    output123 = ssh123.send_command(singleclicommands)
    print(output123)

print("job is success")
```

Script#2

Network Journey
A journey towards packet life !!!

```
from netmiko import ConnectHandler
deviceinfo123 = {
    "device_type": "cisco_ios",
    "ip": "172.16.24.150",
    "username": "admin",
    "password": "cisco",
}
ssh123 = ConnectHandler(**deviceinfo123)
output123 = ssh123.send_config_set(["do show clock", "router ospf 100"])
print(output123)
print("job is success")
```

Script#3

Network Journey
A journey towards packet life !!!

```
from netmiko import ConnectHandler

class CiscoDevice:  #define class
    def __init__(self1, device_type, ip, username, password):  #define methods
        self1.device_info = {
            "device_type": device_type,
            "ip": ip,
            "username": username,
            "password": password
        }
        self1.connection = ConnectHandler(**self1.device_info)

    def send_command(self1, command):  #define methods/functions
        output = self1.connection.send_command(command)
        print(output)

device1 = CiscoDevice("cisco_ios", "172.16.24.177", "admin", "cisco")
 #define object
device1.send_command("show ip int br") #.send_command() is method to call the object
device1.send_command("show clock")
device1.send_command("show run | i hostname")
device1.send_command("show ver")
```

Function:

- print(): A built-in Python function that is used to display output to the console.

Class:

- ConnectHandler: A class from the Netmiko library that is used to create an object to establish an SSH connection to a network device.

Attributes:

- deviceinfo123: A dictionary object that contains the necessary information to connect to the Cisco IOS device, including the device type, IP address, username, and password.

Methods:

- **: The double-asterisk notation is used to pass a dictionary object as a set of keyword arguments to a function or method.
- ConnectHandler(): A method of the ConnectHandler class that creates a new object to establish an SSH connection to a network device.
- send_command(): A method of the ssh123 object that sends a specified command to the connected network device and retrieves the output.

Script#4

In this refactored code, we define a new class called CiscoDevice, which has two methods: **init** and send_command. The **init** method creates a dictionary object with device information and establishes an SSH connection using the ConnectHandler class. The send_command method takes a command argument, sends it to the device using the SSH connection, and prints the resulting output.
We then create a new instance of the CiscoDevice class, device1, and call the send_command method on it to retrieve and print the output of the "show ip int br" command.

Network Journey
A journey towards packet life !!!

```
from netmiko import ConnectHandler

class CiscoDevice:
    def __init__(self, device_type, ip, username, password):
        self.device_info = {
            "device_type": device_type,
            "ip": ip,
            "username": username,
            "password": password
        }
        self.connection = ConnectHandler(**self.device_info)

    def send_commands(self, commands):
        for command in commands:
            output = self.connection.send_command(command)
            print(output)


device1 = CiscoDevice("cisco_ios", "172.16.24.177", "admin", "cisco")

commands = ["show run | i hostname", "show ip int br", "show clock",
"show ver"]

device1.send_commands(commands)

print("job is success")
```

Function:

- print(): A built-in Python function that is used to display output to the console.

Class:

- ConnectHandler: A class from the Netmiko library that is used to create an object to establish an SSH connection to a network device.

Attributes:

- deviceinfo123: A dictionary object that contains the necessary information to connect to the Cisco IOS device, including the device type, IP address, username, and password.

Methods:

- **: The double-asterisk notation is used to pass a dictionary object as a set of keyword arguments to a function or method.
- ConnectHandler(): A method of the ConnectHandler class that creates a new object to establish an SSH connection to a network device.
- send_command(): A method of the ssh123 object that sends a specified command to the connected network device and retrieves the output.

Script#5

In this refactored code, we define a new class called CiscoDevice, which has two methods: init and send_commands. The init method creates a dictionary object with device information and establishes an SSH connection using the ConnectHandler class. The send_commands method takes a list of command arguments, sends each command to the device using the SSH connection, and prints the resulting output.
We then create a new instance of the CiscoDevice class, device1, and call the send_commands method on it with the list of commands to retrieve and print the output for each command. Finally, we print a message indicating the job is successful.

Network Journey
A journey towards packet life !!!

Script#6

```
from netmiko import ConnectHandler
deviceinfo123 = {
    "device_type": "cisco_ios",
    "ip": "172.16.24.205",
    "username": "admin",
    "password": "cisco"
}
ssh123 = ConnectHandler(**deviceinfo123)
mycommands = input('enter the interface id:')
checkinter123 = ssh123.send_command("show ip int " + mycommands)
if "up" and "up" in checkinter123:
    print(mycommands + " is already up")
    y = ssh123.send_config_set(["interface " + mycommands, "shutdown", "desc **reserved for CR12345*"])
    print(y)
else:
    print(mycommands + " is down")
```

Network Journey
A journey towards packet life !!!

Script#7

```
from netmiko import ConnectHandler, NetmikoAuthenticationException, NetmikoTimeoutException
multidevice = ["172.16.24.200", "172.16.24.205"]
for singledevice in multidevice:
    deviceinfo123 = {
        "device_type": "cisco_ios",
        "ip": singledevice,
        "username": "admin",
        "password": "cisco1"
    }
    try:
        ssh123 = ConnectHandler(**deviceinfo123)
        print("taking connection to " + singledevice + "#" * 10)
        out123 = ssh123.send_command("show ip int br")
        print(out123)
    except NetmikoAuthenticationException:
        print(singledevice + " failing due to cred issue")
        takingerrodev = open("authenticationfailed.txt", "a")
        takingerrodev.write(singledevice + "/n")
        takingerrodev.close()
        pass
    except NetmikoTimeoutException:
        print(singledevice + " failing due to TIMEOUT issue")
        takingerrodev = open("Timeoutfailed.txt", "a")
        takingerrodev.write(singledevice + "/n")
        takingerrodev.close()
        pass
print("job completed")
```

Network Journey
A journey towards packet life !!!

Script#7

```
from netmiko import ConnectHandler, NetmikoAuthenticationException, NetmikoTimeoutException



pointingexternalfile = open(r"C:/Users/sagardhawan/Music/deviceip.txt", "r")
ab = pointingexternalfile.readlines()
print(ab)
#
pointingexternalfile1 = open(r"C:/Users/sagardhawan/Music/cred.txt", "r")
ab1 = pointingexternalfile1.readlines()
print(ab1)
user = ab1[0]
pass1 = ab1[1]
#
for singledevice in ab:



 start = 1
 while start < 5:
  try:
    deviceinfo123 = {
      "device_type": "cisco_ios",
      "ip": singledevice,
      "username": user,
      "password": pass1
    }
    ssh123 = ConnectHandler(**deviceinfo123)
    hostname123 = ssh123.find_prompt()
    print(hostname123)
    if ">" or "#" in hostname123:
      print("login to " + singledevice + " is successfully")
      print("taking connection to " + singledevice + "#" * 10)
      out123 = ssh123.send_command("show ip int br")
      print(out123)
      break
    else:
      pass
  except NetmikoAuthenticationException:
    print(singledevice + " failing due to cred issue")
    takingerrodev = open("authenticationfailed.txt", "a")
    takingerrodev.write(singledevice + "/n")
    takingerrodev.close()
    pass
  except NetmikoTimeoutException:
    print(singledevice + " failing due to TIMEOUT issue")
    takingerrodev = open("Timeoutfailed.txt", "a")
    takingerrodev.write(singledevice + "/n")
    takingerrodev.close()
    pass
  print(start)
  start = start + 1
print("job completed")
```

Script#7

```
from netmiko import ConnectHandler, NetmikoAuthenticationException, NetmikoTimeoutException

pointingexternalfile = open(r"C:/Users/sagardhawan/Music/deviceip.txt", "r")
ab = pointingexternalfile.readlines()
print(ab)
#
pointingexternalfile1 = open(r"C:/Users/sagardhawan/Music/cred.txt", "r")
ab1 = pointingexternalfile1.readlines()
print(ab1)
user = ab1[0]
pass1 = ab1[1]
#
for singledevice in ab:

 start = 1
 while start < 5:
  try:
    deviceinfo123 = {
      "device_type": "cisco_ios",
      "ip": singledevice,
      "username": user,
      "password": pass1
    }
    ssh123 = ConnectHandler(**deviceinfo123)
    hostname123 = ssh123.find_prompt()
    print(hostname123)
    if ">" or "#" in hostname123:
      print("login to " + singledevice + " is successfully")
      print("taking connection to " + singledevice + "#" * 10)
      out123 = ssh123.send_command("show ip int br")
      print(out123)
      break
    else:
      pass
  except NetmikoAuthenticationException:
    print(singledevice + " failing due to cred issue")
    takingerrodev = open("authenticationfailed.txt", "a")
    takingerrodev.write(singledevice + "/n")
    takingerrodev.close()
    pass
  except NetmikoTimeoutException:
    print(singledevice + " failing due to TIMEOUT issue")
    takingerrodev = open("Timeoutfailed.txt", "a")
    takingerrodev.write(singledevice + "/n")
    takingerrodev.close()
    pass
  print(start)
  start = start + 1
print("job completed")
```

```
from netmiko import ConnectHandler
from netmiko.ssh_autodetect import SSHDetect

iplist = ["172.16.24.206", "172.16.24.205"]
for ip in iplist:
    device123 = {
        "device_type": "autodetect",
        "ip": ip,
        "username": "admin",
        "password": "cisco"
    }
    guesseer123 = SSHDetect(**device123)
    device_type = guesseer123.autodetect()
    print(device_type)
    if device_type == "cisco_ios":
        device123 = {
            "device_type": device_type,
            "ip": ip,
            "username": "admin",
            "password": "cisco"
        }
        ssh123 = ConnectHandler(**device123)
        0123 = ssh123.send_command("show ip int br")
        print(0123)
    elif device_type == "juniper":
        device123 = {
            "device_type": device_type,
            "ip": ip,
            "username": "admin",
            "password": "cisco"
        }
        ssh123 = ConnectHandler(**device123)
        0123 = ssh123.send_command("set xxxxxx")
        print(0123)
    elif device_type == "cisco_asa":
        device123 = {
            "device_type": device_type,
            "ip": ip,
            "username": "admin",
            "password": "cisco"
        }
        ssh123 = ConnectHandler(**device123)
        0123 = ssh123.send_command("show int ip br")
        print(0123)
    elif device_type == "panos":
        device123 = {
            "device_type": device_type,
            "ip": ip,
            "username": "admin",
            "password": "cisco"
        }
        ssh123 = ConnectHandler(**device123)
        0123 = ssh123.send_command("show int ip br")
        print(0123)
    else:
        print("this is non-ssh based device, excluding " + ip)
```

Script#7

Network Journey
A journey towards packet life !!!

Script#7

```
from netmiko import ConnectHandler, file_transfer
from netmiko.ssh_autodetect import SSHDetect

iplist = ["172.16.24.206", "172.16.24.205"]
for ip in iplist:
    device123 = {
        "device_type": "autodetect",
        "ip": ip,
        "username": "admin",
        "password": "cisco"
    }
    guesseer123 = SSHDetect(**device123)
    device_type = guesseer123.autodetect()
    print(device_type)
    if device_type == "cisco_ios":
        device123 = {
            "device_type": device_type,
            "ip": ip,
            "username": "admin",
            "password": "cisco"
        }
        ssh123 = ConnectHandler(**device123)
        0123 = ssh123.send_command("show ip int br")
        enscp = ssh123.send_config_set("ip scp server enable")
        trnsfer123 = file_transfer(ssh123, source_file="netmiko_global.log", dest_file="ciscoiosimage15.6.log",
                    direction="put", file_system="disko:")
        print(trnsfer123)
        print(0123)
    elif device_type == "juniper":
        device123 = {
            "device_type": device_type,
            "ip": ip,
            "username": "admin",
            "password": "cisco"
        }
        ssh123 = ConnectHandler(**device123)
        0123 = ssh123.send_command("set xxxxxx")
        print(0123)
    elif device_type == "cisco_asa":
        device123 = {
            "device_type": device_type,
            "ip": ip,
            "username": "admin",
            "password": "cisco"
        }
        ssh123 = ConnectHandler(**device123)
        0123 = ssh123.send_command("show int ip br")
        print(0123)
    elif device_type == "panos":
        device123 = {
            "device_type": device_type,
            "ip": ip,
            "username": "admin",
            "password": "cisco"
        }
        ssh123 = ConnectHandler(**device123)
        0123 = ssh123.send_command("show int ip br")
        print(0123)
    else:
        print("this is non-ssh based device, excluding " + ip)
```

==========
HOME WORK
---
1. complete all excer. from 21-april-2023 PDF
slide 112 to 119

2. write a python netmiko script for multiple device (more than 1)
and store the device list externally and execute (show ip int br, show run, show clock) and store the backup in external notepad

3. write a python netmiko script for multiple device (more than 1)
and configure OSPF neighbor and check the neighborship command

reference:
inter <>
ip address <> <>
!
router ospf 100
network 172.16.20.0 0.0.0.255 area 0
!

verify ip ospf neighbor

4. write a python netmiko script for multiple device (more than 1)
and configure BGP neighbor and check the neighborship command

reference:
inter <>
ip address <> <>
!
router bgp 65000
neighbor <> remote-as <>
network 172.16.24.0 255.255.255.0
!

verify the bgp neighborship table

HOMEWORK
21-april-2023

Network Journey
A journey towards packet life !!!

```
from netmiko import ConnectHandler
import xlrd
import re
import csv
import os
workbook123 = xlrd.open_workbook_xls(r"C:/Users/sagardhawan/Downloads/read-from-excel.xls")
sheet123 = workbook123.sheet_by_name("Chennai")
for singledevice in range(1,sheet123.nrows):
    hostname = sheet123.row(singledevice)[1].value
    print(hostname)
    deviceip = sheet123.row(singledevice)[2].value
    print(deviceip)
    devicetype = sheet123.row(singledevice)[3].value
    print(devicetype)
    user123 = sheet123.row(singledevice)[4].value
    print(user123)
    pass123 = sheet123.row(singledevice)[5].value
    print(pass123)
    config123 = sheet123.row(singledevice)[6].value
    listconfig123 = config123.splitlines()
    print(listconfig123)
    #
    deviceinfo123 = {
        "device_type":devicetype,
        "ip": deviceip,
        "username": user123,
        "password": pass123
    }
    ssh123 = ConnectHandler(**deviceinfo123)
    output123 = ssh123.send_command("show ip int br")
    print(output123)
    pushconfig123 = ssh123.send_config_set(listconfig123)
    print(pushconfig123)
```

part-a

Network Journey
A journey towards packet life !!!

```
#capturelogmsgs
openblanknotepad = open(r"C:/Users/sagardhawan/Downloads/backup_" + deviceip + "-" + hostname + ".txt", "w")
copycontent = openblanknotepad.write(pushconfig123)
openblanknotepad.close()
#fetching data using regex
x = ssh123.send_command("show version")
y = ssh123.send_command("show run")
z = ssh123.send_command("show processes memory sorted")
hostname1234 = re.compile("(/S+)/s*uptime/s*is/s*")
findhostname1234 = hostname1234.findall(x)
print(findhostname1234[0])

uptime123 = re.compile("uptime/s*is/s*(.+)")
finduptime123 = uptime123.findall(x)
print(finduptime123[0])

version123 = re.compile("Version/s*(/S+),")
findversion123 = version123.findall(x)
print(findversion123[0])

username123 = re.compile("username/s+(/S+)")
findusername123 = username123.findall(y)
print(findusername123[0])

findtotalmem = re.compile("Total:/s*(/S+)")
findtotalmem123 = findtotalmem.findall(z)
print(findtotalmem123[0])

findfreemem = re.compile("Free:/s*(/S+)")
findfreemem1234 = findfreemem.findall(z)
print(findfreemem1234[0])
```

part-b

```
freememinpercentage = (int(findfreemem1234[0]) / int(findtotalmem123[0])) * 100
print(freememinpercentage)



#
file_exist123 = os.path.isfile(r"C:/Users/sagardhawan/Downloads/fetch_regex.csv")
if file_exist123:
with open(r"C:/Users/sagardhawan/Downloads/fetch_regex.csv", "a", newline="") as csv123:
header123 = ["deviceip", "hostname1234", "finduptime123", "findversion123", "findusername123", "freememinpercentage"]
csv1234 = csv.DictWriter(csv123, fieldnames=header123)
#csv1234.writeheader() # label is printed
csv1234.writerow({"deviceip": deviceip,
"hostname1234": findhostname1234[0],
"finduptime123": finduptime123[0],
"findversion123": findversion123[0],
"findusername123": findusername123[0],
"freememinpercentage":freememinpercentage})
if not file_exist123:
with open(r"C:/Users/sagardhawan/Downloads/fetch_regex.csv", "a", newline="") as csv123:
header123 = ["deviceip", "hostname1234", "finduptime123", "findversion123", "findusername123", "freememinpercentage"]
csv1234 = csv.DictWriter(csv123, fieldnames=header123)
csv1234.writeheader() # label is printed
csv1234.writerow({"deviceip": deviceip,
"hostname1234": findhostname1234[0],
"finduptime123": finduptime123[0],
"findversion123": findversion123[0],
"findusername123": findusername123[0],
"freememinpercentage":freememinpercentage})


print("Job completed")
```

part-c

13. Model Driven Programmability (MDP) Overview – 2hrs
14. Basics of Data Structing Languages (XML, JSON, YAML) – 1hrs
15. Basics of Data Modelling Language (YANG) – 1hrs

Model Driven Programmability (MDP) is an approach to network automation and management that leverages formalized data models to enable more efficient and accurate configuration, operation, and monitoring of network devices and services. This approach has been gaining traction in recent years due to the increasing complexity of networks and the need for more agile and automated management.

Key concepts in MDP include:

1. Data models: MDP relies on structured data models, which provide a standardized way to represent network devices, configurations, and operational data. These models can be defined using modeling languages like YANG, which is widely used in the networking industry.
2. APIs and protocols: MDP uses standardized APIs (Application Programming Interfaces) and protocols to facilitate communication between network devices and management systems. These interfaces enable developers to write applications that can interact with the network without needing deep knowledge of the underlying protocols or hardware. Examples of such protocols include NETCONF, RESTCONF, and gRPC.
3. Abstraction: By providing a high-level, abstracted view of the network, MDP simplifies the task of managing and automating networks. Network engineers and developers can focus on the desired end-state, rather than having to deal with low-level device-specific details.
4. Automation: MDP enables the automation of repetitive tasks, such as configuration changes or software upgrades, which can improve the efficiency and reliability of network operations. Automated processes also reduce the likelihood of human error, leading to more stable networks.
5. Programmability: MDP allows network operators to create custom applications and scripts to manage and monitor their networks. This level of programmability opens up new possibilities for innovation and optimization in network management.
6. Validation: By using formal data models, MDP can automatically validate configurations and operational data, ensuring that they conform to the model's constraints. This helps to reduce configuration errors and improve overall network stability.
7. Interoperability: The use of standardized data models and protocols in MDP promotes interoperability among different network devices and vendors. This makes it easier to integrate and manage multi-vendor networks.

In summary, Model Driven Programmability offers a powerful approach to network management and automation by leveraging structured data models, standardized APIs, and protocols. By abstracting away low-level details and enabling automation, MDP simplifies network operations, reduces human error, and fosters innovation.

Model Driven Programmability (MDP) has been implemented in various ways across the networking industry to improve network management, automation, and monitoring. Here are some examples:

1. OpenDaylight: OpenDaylight is an open-source Software-Defined Networking (SDN) platform that leverages MDP concepts to provide a modular, extensible, and scalable architecture. OpenDaylight uses YANG data models and supports protocols like NETCONF and RESTCONF to enable a consistent and programmable interface for network devices from multiple vendors.
2. Cisco NSO (Network Services Orchestrator): Cisco NSO is an orchestration platform that utilizes MDP principles to automate the provisioning, configuration, and management of multi-vendor network devices. It uses YANG data models and supports NETCONF and RESTCONF protocols to enable seamless interaction between devices and management systems.
3. Juniper Networks' Junos OS: Juniper Networks' Junos operating system supports MDP through its support for YANG data models and the NETCONF protocol. This allows network operators to automate configuration and management tasks, and simplifies integration with third-party tools and systems.
4. OpenConfig: OpenConfig is a collaborative effort among network operators and vendors to develop vendor-neutral, YANG-based data models for network devices and services. By standardizing these data models and supporting protocols like gRPC and NETCONF, OpenConfig promotes interoperability and simplifies network management and automation.
5. Network automation tools and libraries: Various tools and libraries, such as Ansible, NAPALM, and Nornir, have incorporated MDP principles by providing support for YANG data models and protocols like NETCONF, RESTCONF, and gRPC. These tools enable network operators to automate tasks and create custom solutions for network management and monitoring.
6. Intent-Based Networking (IBN): Intent-Based Networking is an approach that uses MDP concepts to translate high-level business objectives into network configurations and policies. IBN platforms like Apstra AOS and Cisco DNA Center use data models and APIs to automate the deployment and management of network services, ensuring that they align with the desired business outcomes.

Network Journey
A journey towards packet life !!!

Netmiko is a popular Python library developed by Kirk Byers, which simplifies the process of connecting to and automating network devices using SSH (Secure Shell). It provides a higher-level API for device interaction and supports a wide range of network devices and vendors.

While Netmiko does not directly adopt Model Driven Programmability (MDP) principles such as YANG data models or standardized protocols like NETCONF and RESTCONF, it does provide a platform for network automation that can be integrated with MDP concepts.

Here are some ways to leverage MDP alongside Netmiko:

1. Custom scripts: You can write custom Python scripts using Netmiko to automate network tasks and integrate them with MDP tools and libraries. For example, you can use Netmiko to gather data from network devices, then process and validate the data using a YANG data model library like pyang or ydk-py.
2. Integration with network automation tools: Netmiko can be used alongside other network automation tools and libraries that support MDP, such as Ansible, NAPALM, and Nornir. These tools can leverage YANG data models, NETCONF, or RESTCONF protocols to interact with network devices, while Netmiko can be used for device-specific or SSH-based automation tasks.
3. Combining with telemetry and monitoring tools: Netmiko can be integrated with telemetry and monitoring tools that adopt MDP concepts, such as tools supporting OpenConfig or gRPC-based streaming telemetry. You can use Netmiko to configure devices for telemetry, while the MDP-based tools handle data collection, processing, and analysis.
4. Vendor-specific extensions: Some network vendors may provide extensions or plugins that enable Netmiko to interact with their devices using MDP principles. These extensions could support YANG data models or other standardized protocols, enhancing the capabilities of Netmiko for specific network devices.

Network Journey
A journey towards packet life !!!

In this example, we will create a Python script using the ncclient library to interact with a network device using the Model Driven Programmability (MDP) principles. The script will connect to the device using the NETCONF protocol, retrieve interface information using a YANG data model, and display the interface details.

1. Install the ncclient library:

bash Copy code

```
pip install ncclient
```

Network Journey
A journey towards packet life !!!

2. Create a Python script (mdp_example.py) with the following content:

```
import xmltodict
from ncclient import manager

# Replace these with your device's credentials and IP address
device_ip = '192.168.1.1'
username = 'your_username'
password = 'your_password'
port = 830  # Default NETCONF port

# Define a function to parse the XML response and display interface information
def display_interfaces(interfaces):
    for interface in interfaces:
        print(f"Interface: {interface['name']}")
        print(f"  Type: {interface['type']}")
        print(f"  Enabled: {interface['enabled']}")
        print(f"  Operational status: {interface['oper-status']}")
        print()

# Connect to the network device using NETCONF
with manager.connect(host=device_ip, port=port, username=username,
                     password=password, hostkey_verify=False) as m:

    # Retrieve interface information using the ietf-interfaces YANG data model
    filter = '''
    <filter>
      <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
        <interface/>
      </interfaces>
    </filter>
    '''

    result = m.get(filter)

# Parse the XML response
interfaces_data = xmltodict.parse(result.xml)["rpc-reply"]["data"]["interfaces"]["interface"]

# Display the interface information
display_interfaces(interfaces_data)
```

Network Journey
A journey towards packet life !!!

3. Replace the 'device_ip', 'username', and 'password' placeholders with your network device's credentials and IP address.
4. Run the script:

bash Copy code

```
python mdp_example.py
```

This script demonstrates the use of Model Driven Programmability (MDP) principles by connecting to a network device using the NETCONF protocol and retrieving interface information based on the ietf-interfaces YANG data model. You can extend this script to perform other tasks, such as configuring interfaces or retrieving other types of data, by modifying the filter and implementing additional functions.

DATA ENCODERS & SERIALIZATION

Network Journey
A journey towards packet life !!!

What are JSON, XML, and YAML?

Text-file formats that can be used to store structured data that can be handy for embedded and Web applications.

XML:

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

<?xml version="1.0" encoding="UTF-8"?>

Example:

<book id="bk101">
<author>Gambardella, Matthew</author>
<title>XML Developer's Guide</title>
<genre>Computer</genre>
<price>44.95</price>
<publish_date>2000-10-01</publish_date>
<description>An in-depth look at creating applications
with XML.</description>
</book>

Network Journey
A journey towards packet life !!!

JSON

JavaScript Object Notation (JSON) is used with JavaScript, of course. It will be familiar to Web developers that use it for client/server communication.

JSON uses name/value pairs

JSON was first standardized in 2013


```
{
"books": [
{
"id": "bk102",
"author": "Crockford, Douglas",
"title": "JavaScript: The Good Parts",
"genre": "Computer",
"price": 29.99,
"publish_date": "2008-05-01",
"description": "Unearthing the Excellence in JavaScript"
}
]
}
```

Network Journey
A journey towards packet life !!!

YAML

YAML stands for YAML Ain’t Markup Language. It uses line and whitespace delimiters instead of explicitly marked blocks that could span one or more lines like XML and JSON. This approach is used in many programming languages, such as Python.

It is commonly used for configuration files and in applications where data is being stored or transmitted.

Example1: Ansible, Flask

Example2:

```
books:
- id: bk102
author: Crockford, Douglas
title: 'JavaScript: The Good Parts'
genre: Computer
price: 29.99
publish_date: !!str 2008-05-01
description: Unearthing the Excellence in JavaScript
```

Application Programming Interface (API)

TYPES OF API:

Open APIs - Open APIs, also known as external or public APIs, are available to developers and other users with minimal restrictions. They may require registration, and use of an API key, or may be completely open. They are intended for external users (developers at other companies, for example) to access data or services. As an example, take a look at the provided by the UK government. Any developer can access it, without even registering, allowing app builders to include governmental data on restaurant standards in their apps.

For example, the traffic app Waze uses public APIs provided by municipalities and other partners about road closures, accidents, construction delays, and service vehicles. In turn, Waze makes cities easier to navigate, which pleases residents and attracts more visitors.

Internal APIs - In contrast to open APIs, internal APIs are designed to be hidden from external users. They are used within a company to share resources. They allow different teams or sections of a business to consume each other's tools, data and programs. Using internal APIs has several advantages over conventional integration techniques, including security and access control, an audit trail of system access, and a standard interface for connecting multiple services.

Partner APIs - Partner APIs are technically similar to open APIs, but they feature restricted access, often controlled through a third-party API gateway. They are usually intended for a specific purpose, such as providing access to a paid-for service. This is a very common pattern in software as a service ecosystem.

For example, Pinterest adopted a submission-based approach to providing access to new data services via its API, requiring partners to submit a request detailing how they would like to use the API before being granted access.

Composite APIs - Composite APIs allow developers to access several endpoints in one call. These could be different endpoints of a single API, or they could be multiple services or data sources. Composite APIs are especially useful in microservice architectures, where a user may need information from several services to perform a single task. Using composite APIs can reduce server load and improve application performance, as one call can return all the data a user needs.

Take this example from Stoplight: Say you want to create an order within a shopping cart API. You might think that this takes just one request. But, in fact, several requests must be made. First, you need to create a customer profile. Then, you need to create the order, add an item, add another, and change the status of the order. Instead of making five separate API calls in succession, you can make just one with a composite API.

Types of API Architectures:

REST - REST (representational state transfer) is a very popular web API architecture. A REST API (or “RESTful” API) is an API that follows REST guidelines and is used for transferring data from a server to a requesting client

l Client-server architecture: the interface is separated from the backend and data storage. This allows for flexibility, and for different components to evolve independent of each other.

l Uniform Interface: All requests and responses must use HTTP as the communication protocol and be formatted in a specific way to ensure compatibility between any client and any server. Server responses are formatted in JavaScript Object Notation (JSON).

l Stateless: Each client-server interaction is independent of every other interaction. The server stores no data from client requests and remembers nothing from past interactions.Cacheability: clients can cache responses, so a REST API response must explicitly state whether it can be cached or not.

l Layered system: Requests and responses must always be formatted the same way, even when passed through intermediate servers between the client and the API.

l Cacheable: Server responses should indicate whether a provided resource can be cached by the client and for how long.

By following these guidelines, REST APIs can be used for quick, easy, secure data transfers, making them a popular choice among developers.

SOAP - SOAP (simple object access protocol) is a protocol for transmitting data across networks and can be used to build APIs. SOAP is standardized by the World Wide Web Consortium (W3C) and utilizes XML to encode information.

SOAP strictly defines how messages should be sent and what must be included in them. This makes SOAP APIs more secure than REST APIs, although the rigid guidelines also make them more code-heavy and harder to implement in general.

For this reason, SOAP is often implemented for internal data transfers that require high security, and the more flexible REST architecture is deployed more commonly everywhere else. But, one more advantage to SOAP is that it works over any communication protocol (not just HTTP, as is the case with REST).

JSON-RPC and XML-RPC - The RPC (Remote Procedural Call) protocol is the most straightforward of the three architectures. Unlike REST and SOAP that facilitate the transfer of data, RPC APIs invoke processes. In other words, they execute scripts on a server.

RPC APIs may employ either JSON (a JSON-RPC protocol) or XML (an XML-RPC protocol) in their calls. XML is more secure and more accommodating than JSON, but these two protocols are otherwise similar. Though the RPC protocol is strict, it's a relatively simple and easy way to execute code on remote networks.

RPC APIs are limited in their security and capabilities, so you likely won’t see them as often as REST or SOAP APIs on the web. However, it can be used for internal systems for making basic process requests, especially many at once.

Representational State Transfer (REST) APIs

An API that uses REST is often referred to a RESTful API. RESTful APIs use HTTP methods to gather and manipulate data. Because there is a defined structure for how HTTP works, it offers a consistent way to interact with APIs from multiple vendors. REST uses different HTTP functions to interact with the data.

HTTP Function	Action	Use Case
GET	Requests data from a destination	Viewing a website
POST	Submits data to a specific destination	Submitting login credentials
PUT	Replaces data in a specific destination	Updating an NTP server
PATCH	Appends data to a specific destination	Adding an NTP server
DELETE	Removes data from a specific destination	Removing an NTP server

HTTP functions are similar to the functions that most applications or databases use to store or alter data —whether the data is stored in a database or within the application. These functions are called “CRUD” functions.

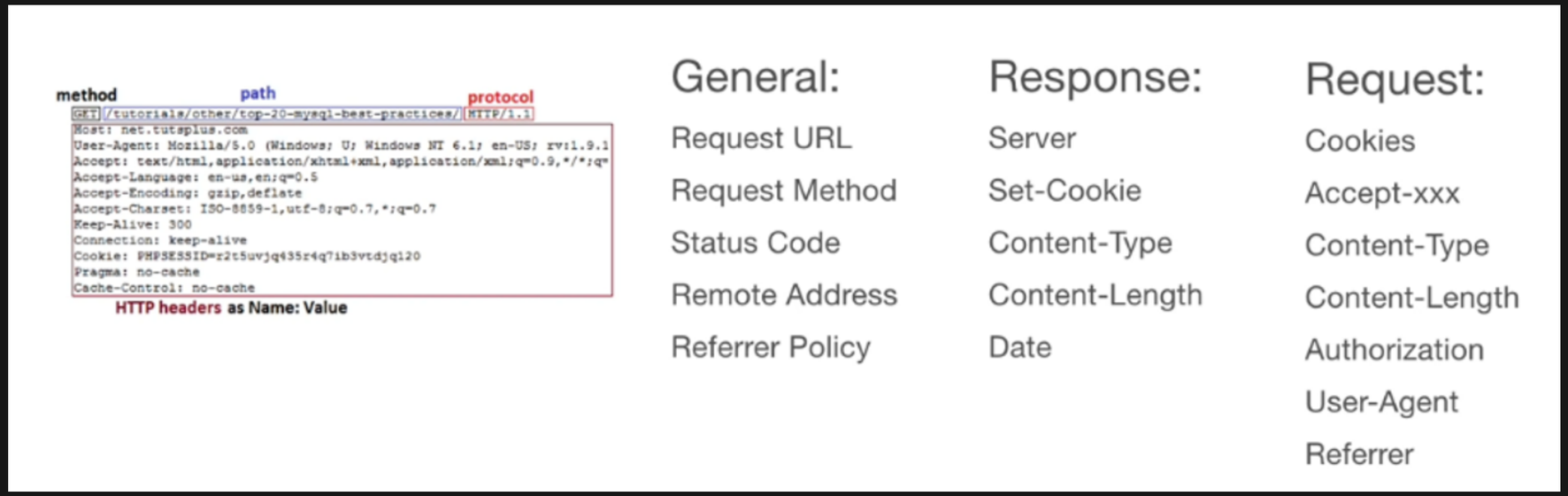
CRUD and REST are two of the most popular concepts in the Application Program Interface (API) industry. REST was made to standardize the HTTP protocol interface between clients and servers and is one of the widely used design styles for web API. On the other hand, CRUD is an acronym used to refer to the four basic operations executed on database applications.

Network Journey
A journey towards packet life !!!

HTTP Headers

The HTTP headers and parameters provide of a lot of information that can help you trace issues when you encounter them. HTTP headers are an essential part of an API request and response as they represent the metadata associated with the API request and response. Headers carry information for the following:

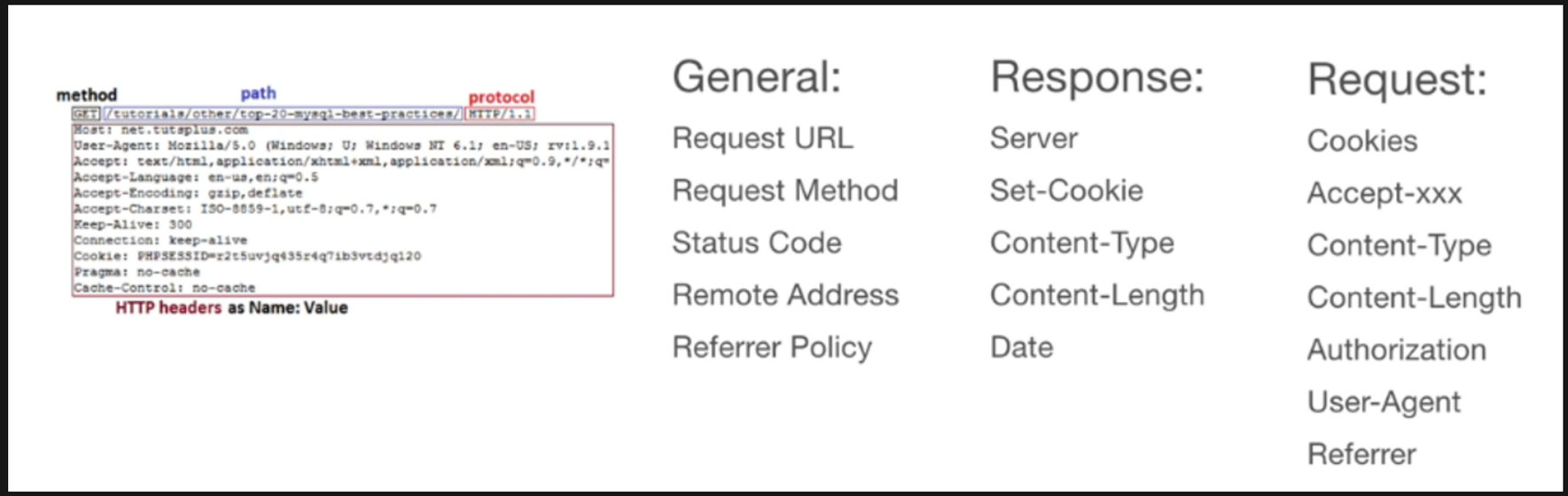
- Request and response body
- Request authorization
- Response caching
- Response cookies



HTTP Headers

The HTTP headers and parameters provide of a lot of information that can help you trace issues when you encounter them. HTTP headers are an essential part of an API request and response as they represent the metadata associated with the API request and response. Headers carry information for the following:

- Request and response body
- Request authorization
- Response caching
- Response cookies



Request Headers

The request headers appear as name:value pairs. Multiple values, separated by commas, can be specified as follows:

request-header-name: request-header-value1, request-header-value2, ...

The following are some examples of request headers:

Host: myhouse.cisco.com

Connection: Keep-Alive

Accept: image/gif, image/jpeg, */*

Accept-Language: us-en, fr, cn

Network Journey
A journey towards packet life !!!

Response Headers

The response headers also appear as name:value pairs. As with request headers, multiple values can be specified as follows:

response-header-name: response-header-value1, response-header-value2, ...

The following are some examples of response headers:

Content-Type: text/html

Content-Length: 35

Connection: Keep-Alive

Keep-Alive: timeout=15, max=100

The response message body contains the resource data requested.

The following are some examples of request and response headers:

·Authorization: Carries credentials containing the authentication information of the client for the resource being requested.

·WWW-Authenticate: This is sent by the server if it needs a form of authentication before it can respond with the actual resource being requested. It is often sent along with response code 401, which means “unauthorized.”

·Accept-Charset: This request header tells the server which character sets are acceptable by the client.

·Content-Type: This header indicates the media type (text/HTML or application/JSON) of the client request sent to the server by the client, which helps process the request body correctly.

·Cache-Control: This header is the cache policy defined by the server. For this response, a cached response can be stored by the client and reused until the time defined in the Cache-Control header.

Response Codes

The first line of a response message (that is, the status line) contains the response status code, which the server generates to indicate the outcome of the request. Each status code is a three-digit number:

- 1xx (informational): The request was successfully received; the server is continuing the process.
- 2xx (success): The request was successfully received, understood, accepted, and serviced.
- 3xx (redirection): Further action must be taken to complete the request.
- 4xx (client error): The request cannot be understood or is unauthorized or the requested resource could not be found.
- 5xx (server error): The server failed to fulfill a request.

Network Journey
A journey towards packet life !!!

Introduction to Postman

Interaction with a software controller using RESTful APIs.

It also discussed being able to test code to see if the desired outcomes are accomplished when executing the code.

Keep in mind that APIs are software interfaces into an application or a controller.

Many APIs require authentication.

This means that such an API is considered just like any other device to which a user needs to authenticate to gain access to utilize the APIs.

A developer who is authenticated has access to making changes using the API, which can impact that application.

This means if a REST API call is used to delete data, that data will be removed from the application or controller just as if a user logged into the device via the CLI and deleted it.

It is best practice to use a test lab or the Cisco DevNet sandbox while learning or practicing any of these concepts to avoid accidental impact to a production or lab environment.

The Postman application has various sections that you can interact with. The focus here is on using the Builder portion of the dashboard. The following sections are the ones that require the most focus and attention:

- History
- Collections
- New Tab
- URL bar

Network Journey
A journey towards packet life !!!

Network Journey
A journey towards packet life !!!

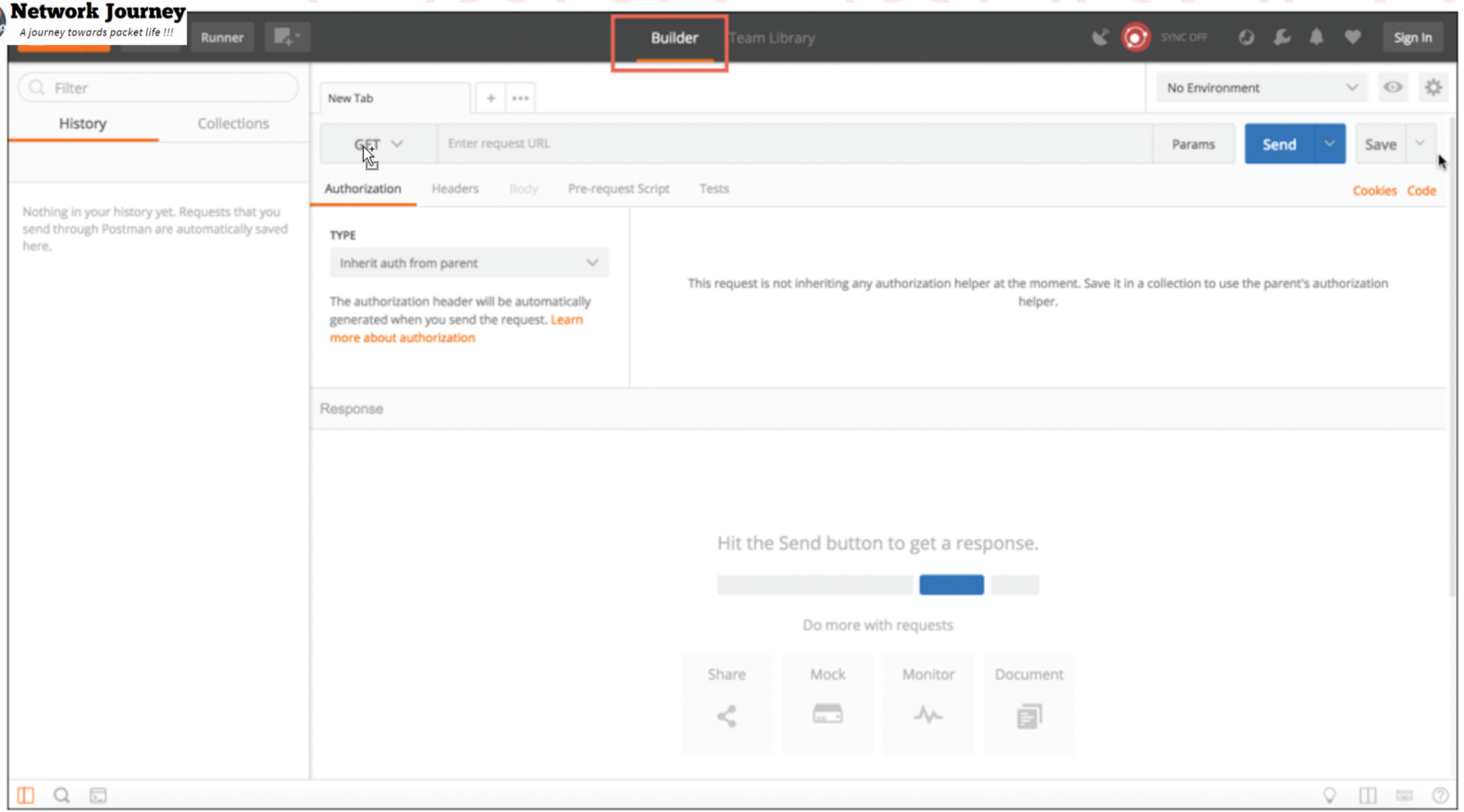


Figure 28-2 *Postman Dashboard*

Data Models and Supporting Protocols

1. Yet Another Next Generation (YANG) modeling language
2. Network Configuration Protocol (NETCONF)
3. RESTCONF

Data Models and Supporting Protocols

1. Yet Another Next Generation (YANG) modeling language
2. Network Configuration Protocol (NETCONF)
3. RESTCONF

Data models describe the things you can configure, monitor, and the actions you can perform on a network device.

Yet Another Next Generation (YANG) modeling language

YANG(Yet Another Next Generation) is a data modeling language for the definition of data sent over network management protocols such as the NETCONF and RESTCONF.
Many network management protocols have associated data modeling languages. SNMP is widely used for fault handling and monitoring. However, it is not often used for configuration changes. CLI scripting is used more often than other methods. The data modeling language associated with SNMP was called the Structure of Management Information (SMI). In the late 1990s, a project was started to create a replacement for SMIv2, which was called SMIng, which was failed.

YANG data models are an alternative to SNMP MIBs and are becoming the standard for data definition languages.

YANG, which is defined in RFC 6020, uses data models. Data models are used to describe whatever can be configured on a device, everything that can be monitored on a device, and all the administrative actions that can be executed on a device, such as resetting counters or rebooting the device. This includes all the notifications that the device is capable of generating. All these variables can be represented within a YANG model. Data models are very powerful in that they create a uniform way to describe data, which can be beneficial across vendors' platforms.

Data models allow network operators to configure, monitor, and interact with network devices holistically across the entire enterprise environment.

YANG models use a tree structure. Within that structure, the models are similar in format to XML and are constructed in modules. These modules are hierarchical in nature and contain all the different data and types that make up a YANG device model. YANG models make a clear distinction between configuration data and state information. The tree structure represents how to reach a specific element of the model, and the elements can be either configurable or not configurable.
Every element has a defined type. For example, an interface can be configured to be on or off. However, the operational interface state cannot be changed; for example, if the options are only up or down, it is either up or down, and nothing else is possible.

NETCONF

NETCONF, defined in RFC 4741 and RFC 6241, is an IETF (Internet Engineering Task Force) standard protocol that uses the YANG data models to communicate with the various devices on the network.

NETCONF runs over SSH, TLS, and (although not common), Simple Object Access Protocol (SOAP).

One of the most important differences is that SNMP can’t distinguish between configuration data and operational data, but NETCONF can.

The following is a list of some of the common use cases for NETCONF:

- Collecting the status of specific fields
- Changing the configuration of specific fields
- Taking administrative actions
- Sending event notifications
- Backing up and restoring configurations
- Testing configurations before finalizing the transaction

Network Journey
A journey towards packet life !!!

Table 28-6 Differences Between SNMP and NETCONF

Feature	SNMP	NETCONF
Resources	OIDs	Paths
Data models	Defined in MIBs	YANG core models
Data modeling language	SMI	YANG
Management operations	SNMP	NETCONF
Encoding	BER	XML, JSON
Transport stack	UDP	SSH/TCP

RESTCONF

RESTCONF, defined in RFC 8040, is used to programmatically interface with data defined in YANG models while also using the datastore concepts defined in NETCONF.

There is a common misconception that RESTCONF is meant to replace NETCONF, but this is not the case. Both are very common methods used for programmability and data manipulation. If fact, RESTCONF uses the same YANG models as NETCONF and Cisco IOS XE.

The goal of RESTCONF is to provide a RESTful API experience while still leveraging the device abstraction capabilities provided by NETCONF.

RESTCONF supports the following HTTP methods and CRUD operations:

- GET
- POST
- PUT
- DELETE
- OPTIONS

Network Journey
A journey towards packet life !!!

Network Journey
A journey towards packet life !!!

PROJECTS FOR 26 APRIL to 4 MAY

1. Network Device Inventory: Write a script that connects to multiple network devices using Netmiko and collects information about each device, such as its hostname, IP address, interface information, and VLAN information. Store the collected information in a CSV or Excel file, which can be used to create a network device inventory.

2. Configuration Backup: Write a script that connects to network devices using Netmiko and backs up their configurations to a file. The script should prompt the user for the device IP address, username, password, and backup file location. It should then use Netmiko to connect to the device and retrieve its configuration, which should be written to the backup file.

3. VLAN Configuration: Write a script that connects to a network device using Netmiko and configures VLANs on the device. The script should prompt the user for the device IP address, username, and password, as well as the VLAN configuration details, such as VLAN ID, name, and interface assignments. The script should then use Netmiko to connect to the device and configure the VLANs.

4. Network Health Check: Write a script that connects to multiple network devices using Netmiko and performs a health check on the network. The health check can include tasks such as verifying device connectivity, checking interface status, testing network performance, and identifying potential issues. The script should output the results of the health check to a file or email them to a specified recipient.

5. Automated Network Troubleshooting: Write a script that connects to a network device using Netmiko and performs automated troubleshooting based on predefined rules. For example, the script could detect an interface that is down and automatically attempt to bring it back up, or it could detect a configuration error and automatically correct it. The script should output the results of the troubleshooting to a file or email them to a specified recipient.

SOLUTIONS

SOLUTIONS #1


```
import csv
from netmiko import ConnectHandler

# Create a list of devices to connect to
devices = [
    {
        "device_type": "cisco_ios",
        "ip": "192.168.1.1",
        "username": "admin",
        "password": "password",
    },
    {
        "device_type": "cisco_ios",
        "ip": "192.168.1.2",
        "username": "admin",
        "password": "password",
    },
    # Add more devices as needed
]

# Open a CSV file to store the collected information
with open("network_device_inventory.csv", "w", newline="") as csv_file:
    csv_writer = csv.writer(csv_file)
    # Write headers to the CSV file
    csv_writer.writerow(["Hostname", "IP Address", "Interfaces", "VLANs"])
    # Connect to each device and collect information
    for device in devices:
        with ConnectHandler(**device) as net_connect:
            hostname = net_connect.send_command("show run | i hostname").split()[-1]
            ip_address = device["ip"]
            interfaces = net_connect.send_command("show ip interface brief")
            vlans = net_connect.send_command("show vlan brief")
        # Write the collected information to the CSV file
        csv_writer.writerow([hostname, ip_address, interfaces, vlans])
```

SOLUTIONS #2


```
from netmiko import ConnectHandler

# Prompt the user for device information and backup file location
ip_address = input("Enter device IP address: ")
username = input("Enter device username: ")
password = input("Enter device password: ")
backup_file = input("Enter backup file location: ")

# Connect to the device using Netmiko and backup its configuration
device = {
    "device_type": "cisco_ios",
    "ip": ip_address,
    "username": username,
    "password": password,
}
with ConnectHandler(**device) as net_connect:
    backup_config = net_connect.send_command("show run")
# Write the backup configuration to the specified file
with open(backup_file, "w") as backup_file:
    backup_file.write(backup_config)
```

SOLUTIONS #3


```
from netmiko import ConnectHandler

# Prompt the user for device information and VLAN configuration details
ip_address = input("Enter device IP address: ")
username = input("Enter device username: ")
password = input("Enter device password: ")
vlan_id = input("Enter VLAN ID: ")
vlan_name = input("Enter VLAN name: ")
interface_list = input("Enter interface list (comma-separated): ")

# Connect to the device using Netmiko and configure the VLAN
device = {
    "device_type": "cisco_ios",
    "ip": ip_address,
    "username": username,
    "password": password,
}
with ConnectHandler(**device) as net_connect:
    # Create the VLAN
    vlan_commands = [f"vlan {vlan_id}", f"name {vlan_name}"]
    net_connect.send_config_set(vlan_commands)
    # Assign interfaces to the VLAN
    interface_commands = [f"interface {interface}" for interface in interface_list.split(",")]
    interface_commands.append(f"switchport access vlan {vlan_id}")
    net_connect.send_config_set(interface_commands)
```

SOLUTIONS #4


```
import csv
from netmiko import ConnectHandler

# Create a list of devices to connect to
devices = [
    {
        "device_type": "cisco_ios",
        "ip": "192.168.1.1",
        "username": "admin",
        "password": "password",
    },
    {
        "device_type": "cisco_ios",
        "ip": "192.168.1.2",
        "username": "admin",
        "password": "password",
    },
    # Add more devices as needed
]

# Open a CSV file to store the collected information
with open("network_health_check.csv", "w", newline="") as csv_file:
    csv_writer = csv.writer(csv_file)
    # Write headers to the CSV file
    csv_writer.writerow(["Device", "Connectivity", "Interface Status", "Network Performance", "Issues"])
    # Connect to each device and perform health check
    for device in devices:
        with ConnectHandler(**device) as net_connect:
            # Check device connectivity
            try:
                ping_output = net_connect.send_command("ping 8.8.8.8")
                connectivity = "Success" if "!" in ping_output else "Failed"
            except:
                connectivity = "Failed"
            # Check interface status
            interface_status = net_connect.send_command("show ip interface brief")
            # Test network performance
            network_performance = net_connect.send_command("ping 192.168.1.1 repeat 100")
            # Identify potential issues
            issues = ""
            if "down" in interface_status:
                issues += "Interface Down; "
            if "Success rate is 0 percent" in network_performance:
                issues += "Network Performance Issues; "
        # Write the collected information to the CSV file
        csv_writer.writerow([device["ip"], connectivity, interface_status, network_performance, issues])
```

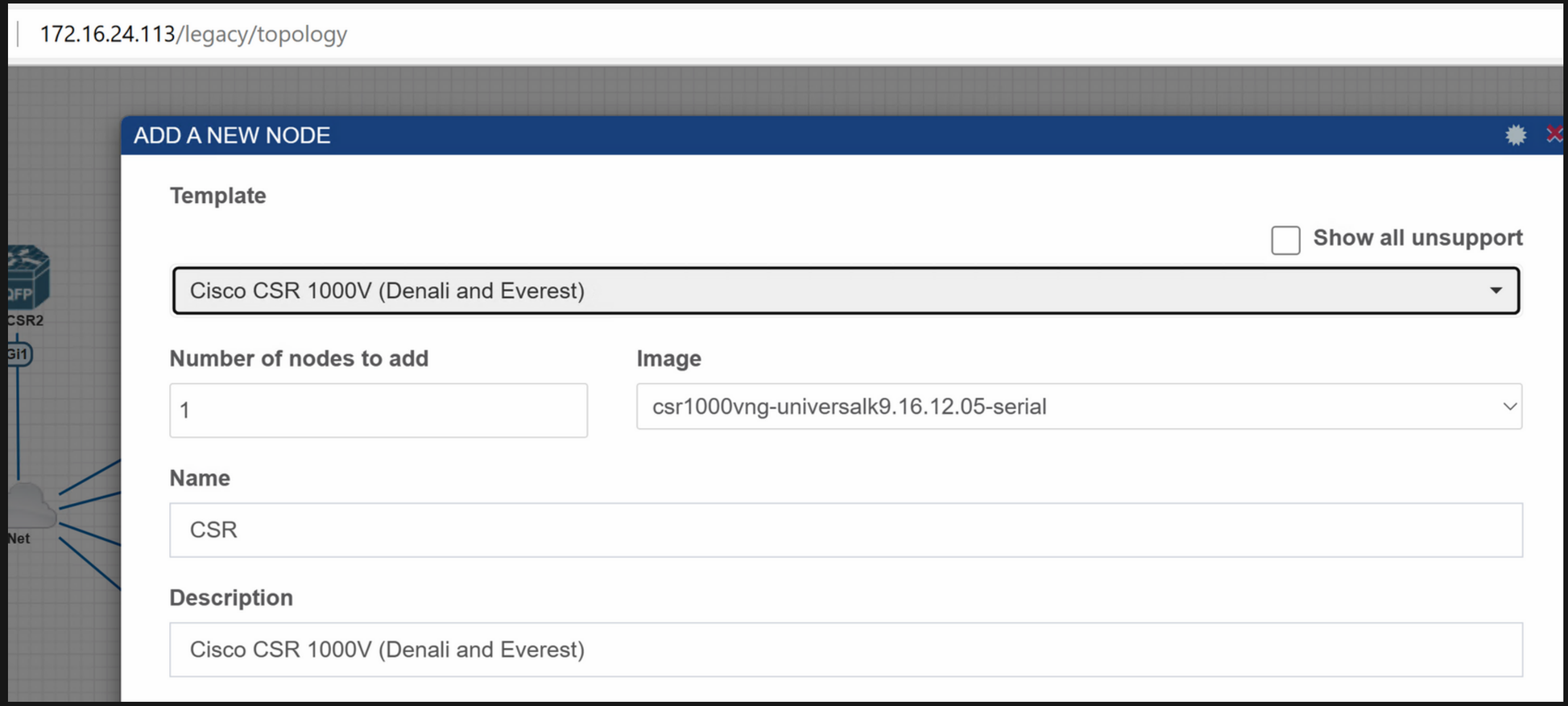
SOLUTIONS #5


```
from netmiko import ConnectHandler

# Prompt the user for device information and troubleshooting rules
ip_address = input("Enter device IP address: ")
username = input("Enter device username: ")
password = input("Enter device password: ")
interface_down_action = input("Enter action for down interfaces (up, shut): ")
config_error_action = input("Enter action for configuration errors (fix, ignore): ")

# Connect to the device using Netmiko and perform troubleshooting
device = {
    "device_type": "cisco_ios",
    "ip": ip_address,
    "username": username,
    "password": password,
}
with ConnectHandler(**device) as net_connect:
    # Check for down interfaces
    down_interfaces = net_connect.send_command("show ip interface brief | i down")
    if down_interfaces:
        if interface_down_action == "up":
            up_commands = [f"interface {interface}", "no shutdown"]
            for interface in down_interfaces.splitlines():
                net_connect.send_config_set(up_commands)
        elif interface_down_action == "shut":
            shut_commands = [f"interface {interface}", "shutdown"]
            for interface in down_interfaces.splitlines():
                net_connect.send_config_set(shut_commands)
    # Check for configuration errors
    config_errors = net_connect.send_command("show running-config | include ^!|\\s{2,}")
    if config_errors:
        if config_error_action == "fix":
            config_commands = [line.strip() for line in config_errors.splitlines() if not line.startswith("!")]
            net_connect.send_config_set(config_commands)
        elif config_error_action == "ignore":
            pass
```

CISCO IOS-XE API AUTOMATION

Network Journey
A journey towards packet life !!!

STEP 1 : GOTO LAB TOPOLOGY AND ADD LASTST CSR 1000 KV ROUTER (support RESTCONF on 16.x and above versions)



Network Journey
A journey towards packet life !!!

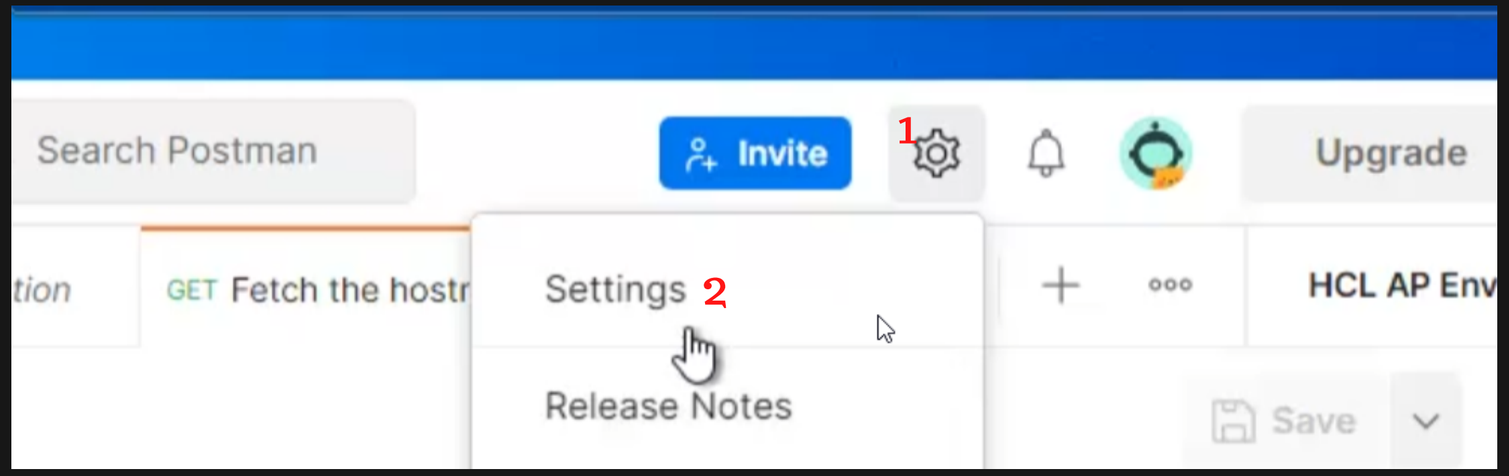
STEP2: DO THE MANAGEMENT CONFIG AS DONE PREVIOUSLY AND ALSO ENABLE "HTTP", "HTTPS" and "RESTCONF"

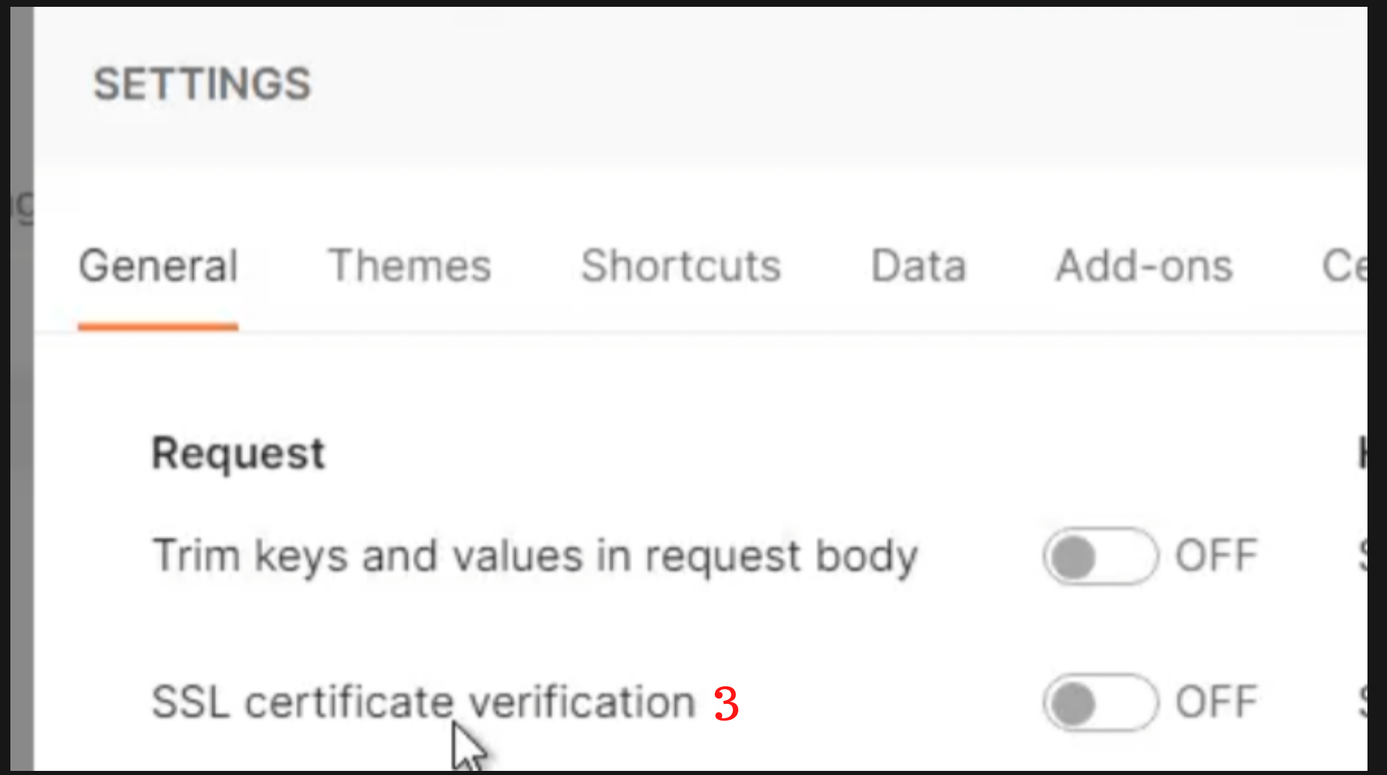
ADDITIONAL COMMANDS NEEDED WHILE TESTING API ON CISCO IOS-XE

```
!
conf terminal
ip http server
ip http secure-server
restconf
!
end
wr mem
!
```

STEP3: GOTO POSTMAN TOOL > SETTINGS > DISABLE SSL CERTIFICATE VERIFICATION : OFF

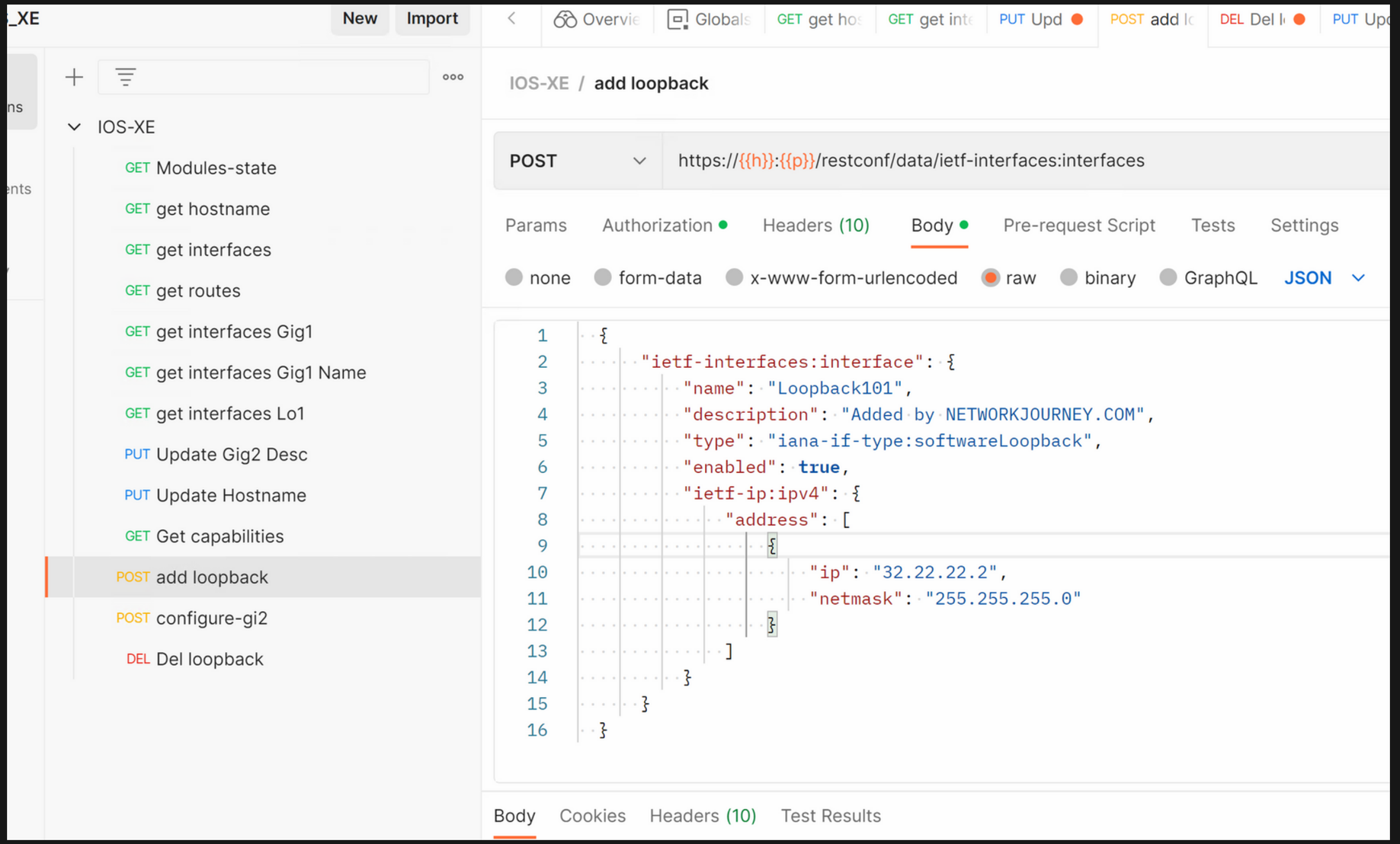
Network Journey
A journey towards packet life !!!





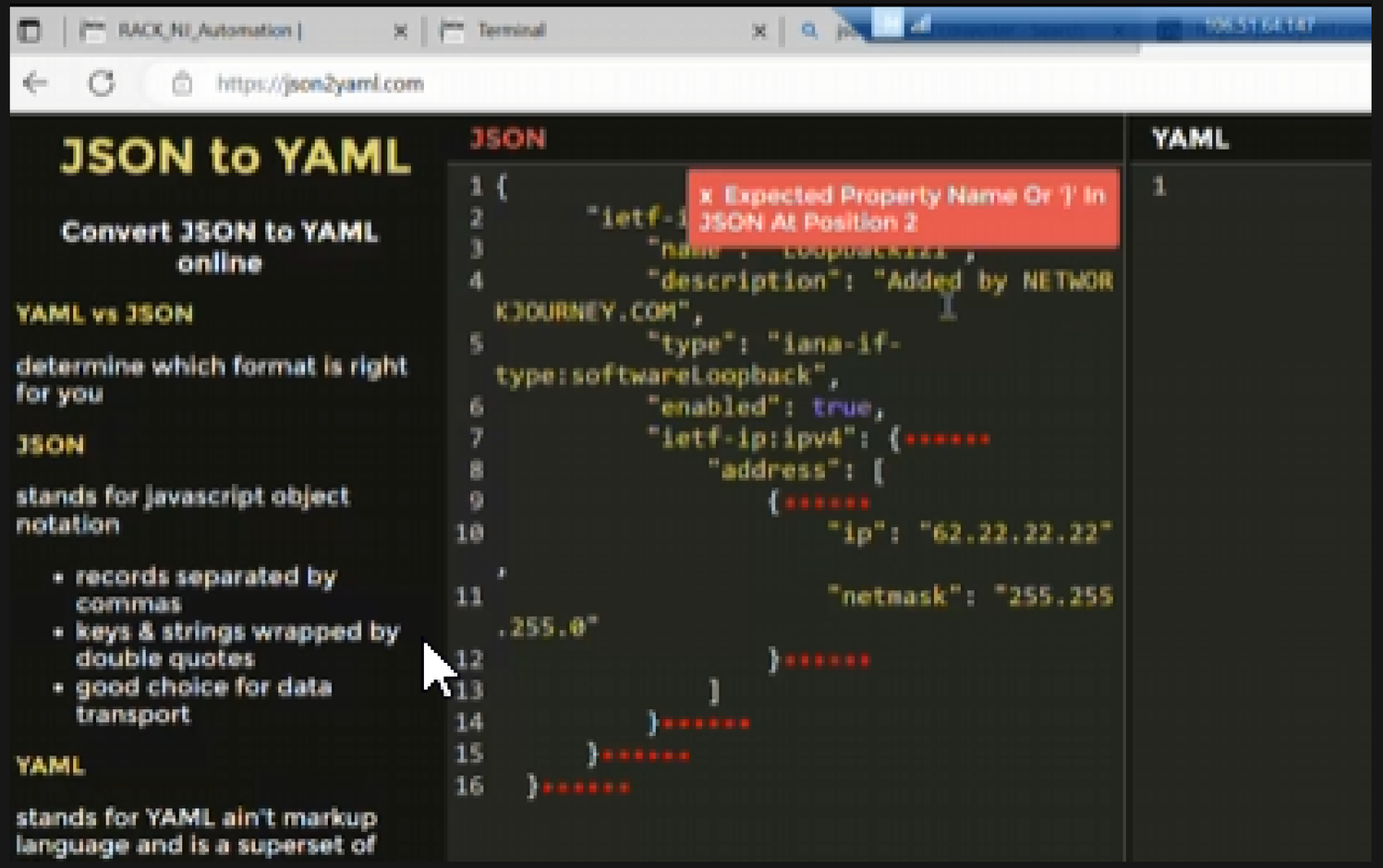
Network Journey
A journey towards packet life !!!

STEP4: GOTO POSTMAN TOOL AND START WORKING THE JSON BACKUP FILE IS GIVEN TO PARTICIPANTS



Network Journey
A journey towards packet life !!!

FREQUENT SEEN ERRORS
: HEADER MODIFICATION DEPENDING UPON ERRORS



Download Payload for creating loopback:

https://drive.google.com/file/d/1UojDFAkLLaYY4QIhfH6KVabi11WE-QdN/view?usp=share_link

Use JSON2YAML.COM online convertor and fix the body > raw > json value

Network Journey
A journey towards packet life !!!

FREQUENT SEEN ERRORS
: HEADER MODIFICATION DEPENDING UPON ERRORS

GET request > HEADER > Add below entry

☑	Accept	application/yang-data+json

POST request > HEADER > Add below entry

☑	Content-Type	application/yang-data+json

PATH JSON FILE

https://github.com/jeremycohoe/cisco-ios-xe-postman-collections/blob/master/cisco_ios_xe_17_2_collection.json

ORIGINAL COPY FOR CISCO IOS-XE POSTMAN COLLECTION

https://drive.google.com/file/d/1N1uTd6CeYDVQ4ZiMi2AafyZASPjTfFSB/view?usp=share_link

Network Journey
A journey towards packet life !!!

Introduction to POSTMAN COMMUNITY TOOL

Interaction with a software controller using RESTful APIs.
It also discussed being able to test code to see if the desired outcomes are accomplished when executing the code.

Keep in mind that APIs are software interfaces into an application or a controller.
Many APIs require authentication.

This means that such an API is considered just like any other device to which a user needs to authenticate to gain access to utilize the APIs.

A developer who is authenticated has access to making changes using the API, which can impact that application.

This means if a REST API call is used to delete data, that data will be removed from the application or controller just as if a user logged into the device via the CLI and deleted it.
It is best practice to use a test lab or the Cisco DevNet sandbox while learning or practicing any of these concepts to avoid accidental impact to a production or lab environment.

The Postman application has various sections that you can interact with. The focus here is on using the Builder portion of the dashboard. The following sections are the ones that require the most focus and attention:

- History
- Collections
- New Tab
- URL bar

Network Journey
A journey towards packet life !!!

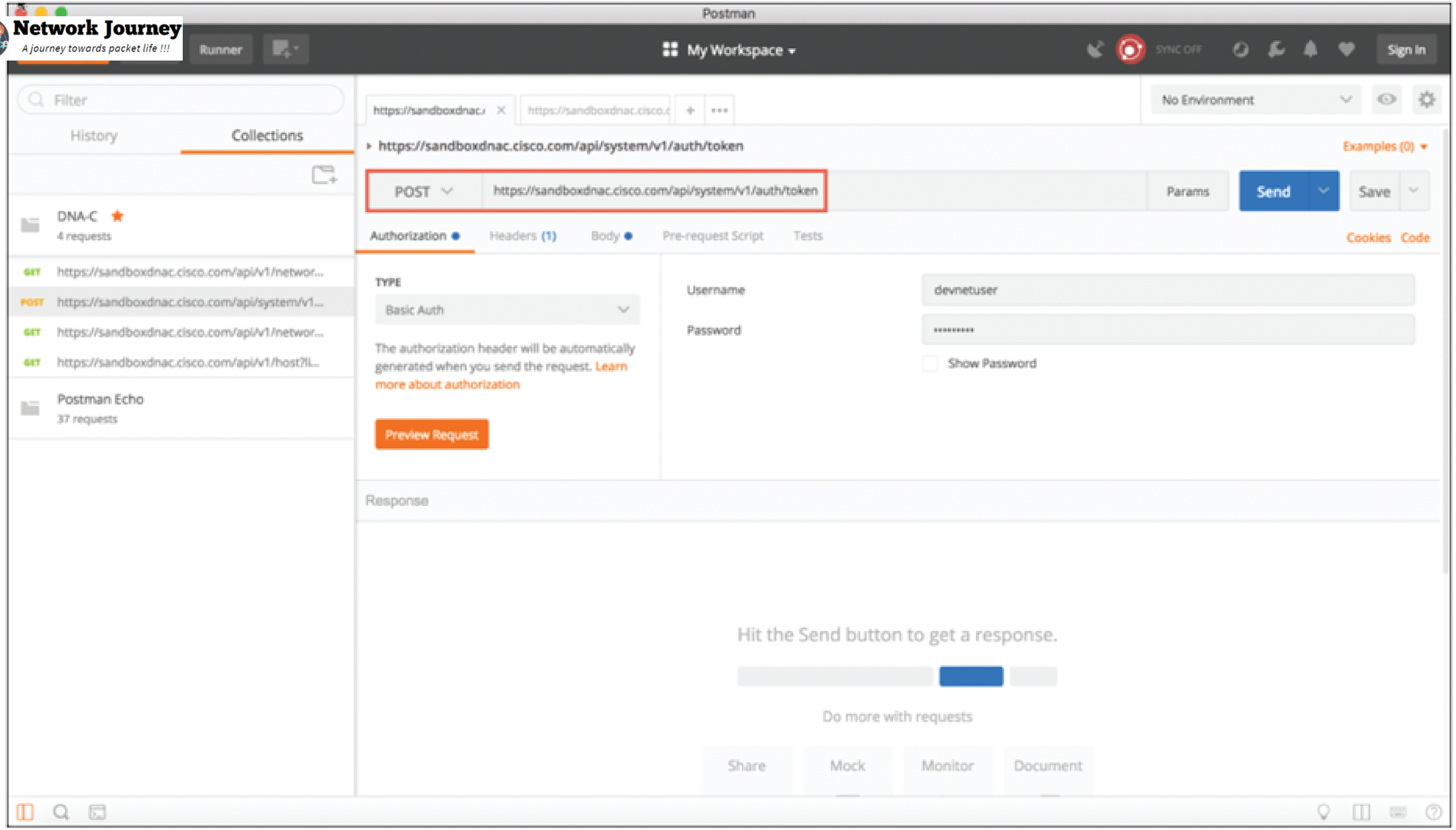


Figure 28-5 *Postman URL Bar with Cisco DNA Center Token API Call*

Network Journey
A journey towards packet life !!!

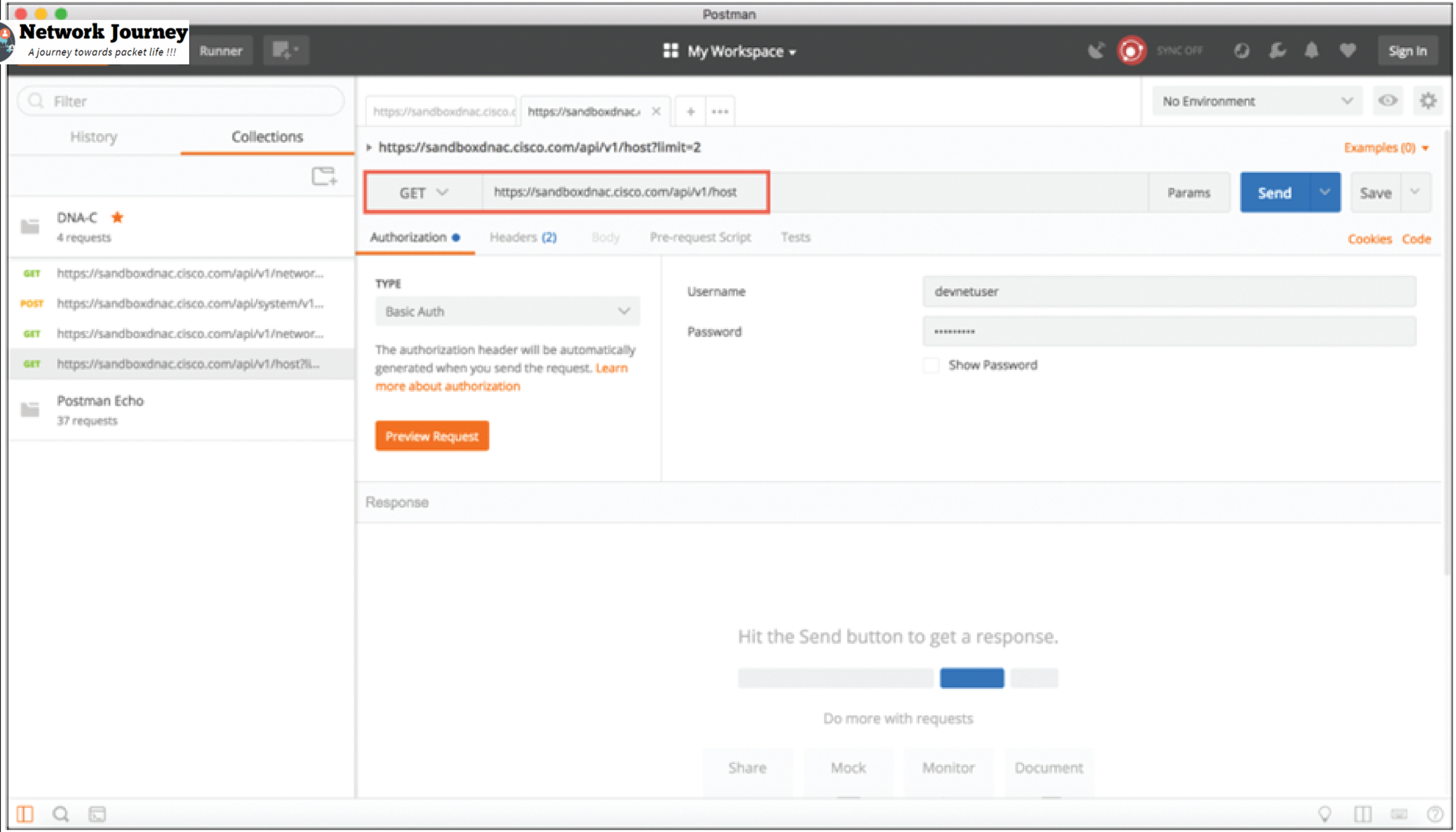


Figure 28-6 *Postman URL Bar with Cisco DNA Center Host API Call*

Network Journey
A journey towards packet life !!!

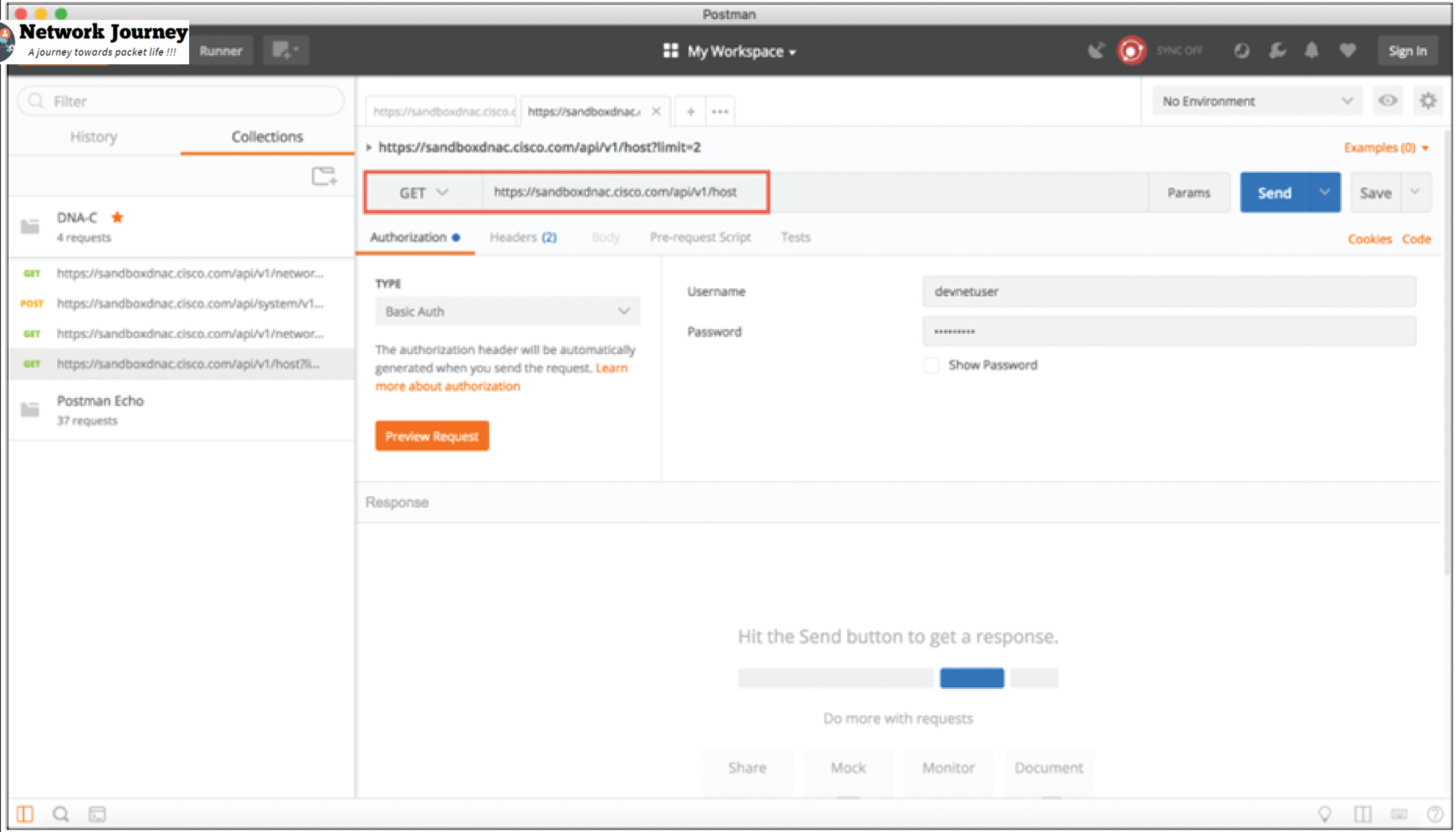


Figure 28-6 *Postman URL Bar with Cisco DNA Center Host API Call*

Network Journey
A journey towards packet life !!!

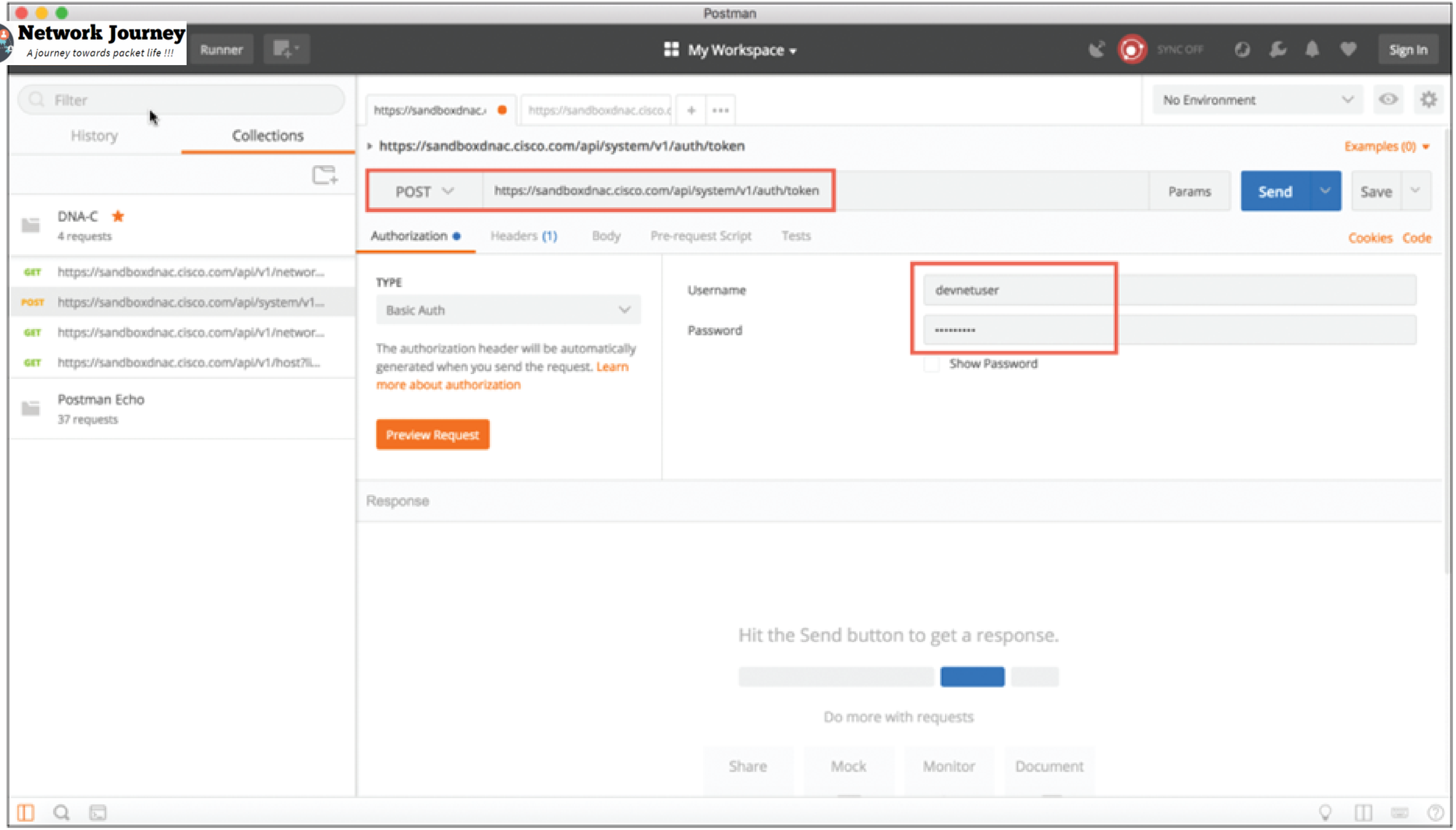


Figure 28-7 *Setting Up Postman to Authenticate with the Cisco DNA Center Controller*

Network Journey
A journey towards packet life !!!

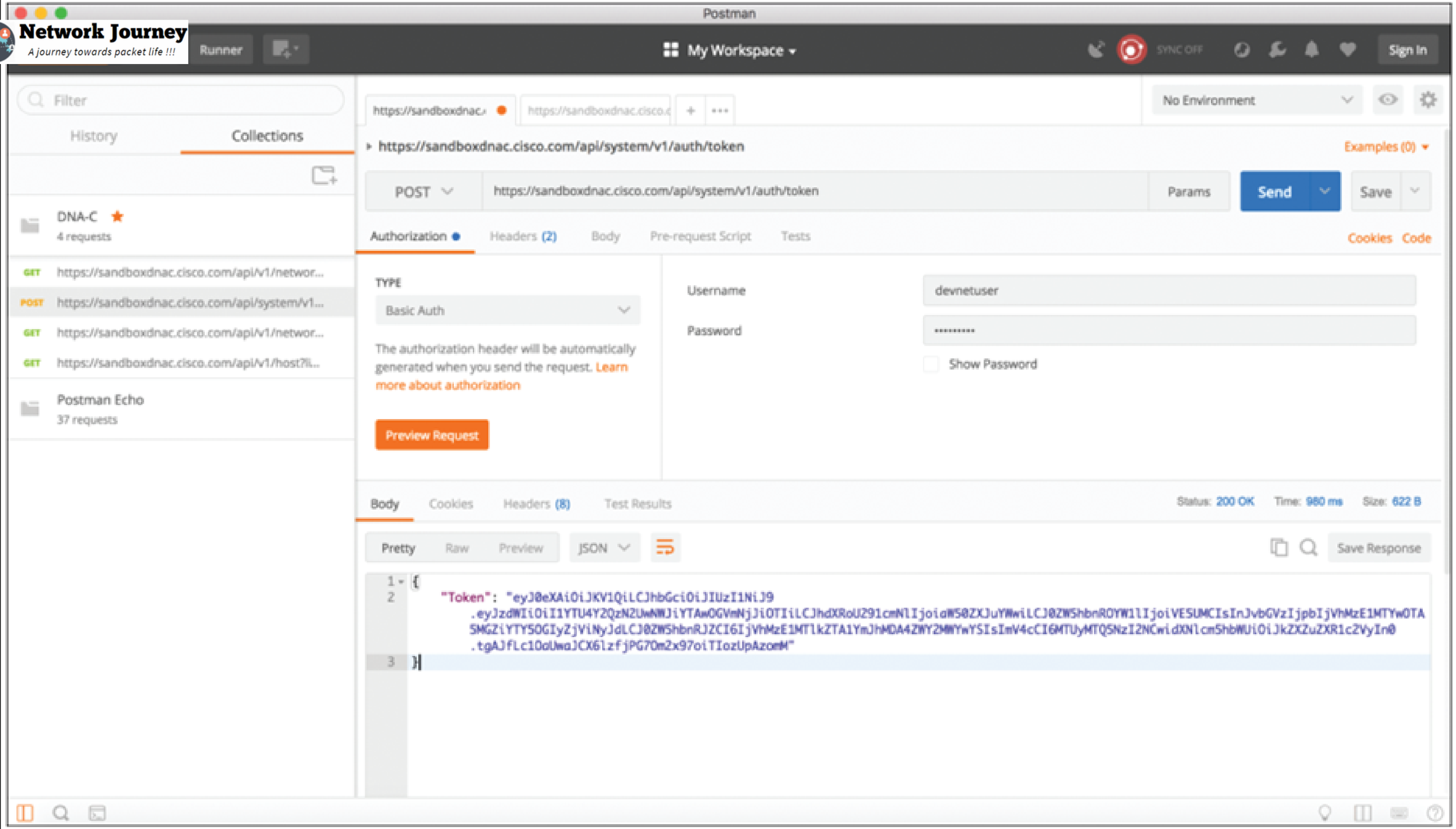


Figure 28-8 *Cisco DNA Center POST Operation*

Network Journey
A journey towards packet life !!!

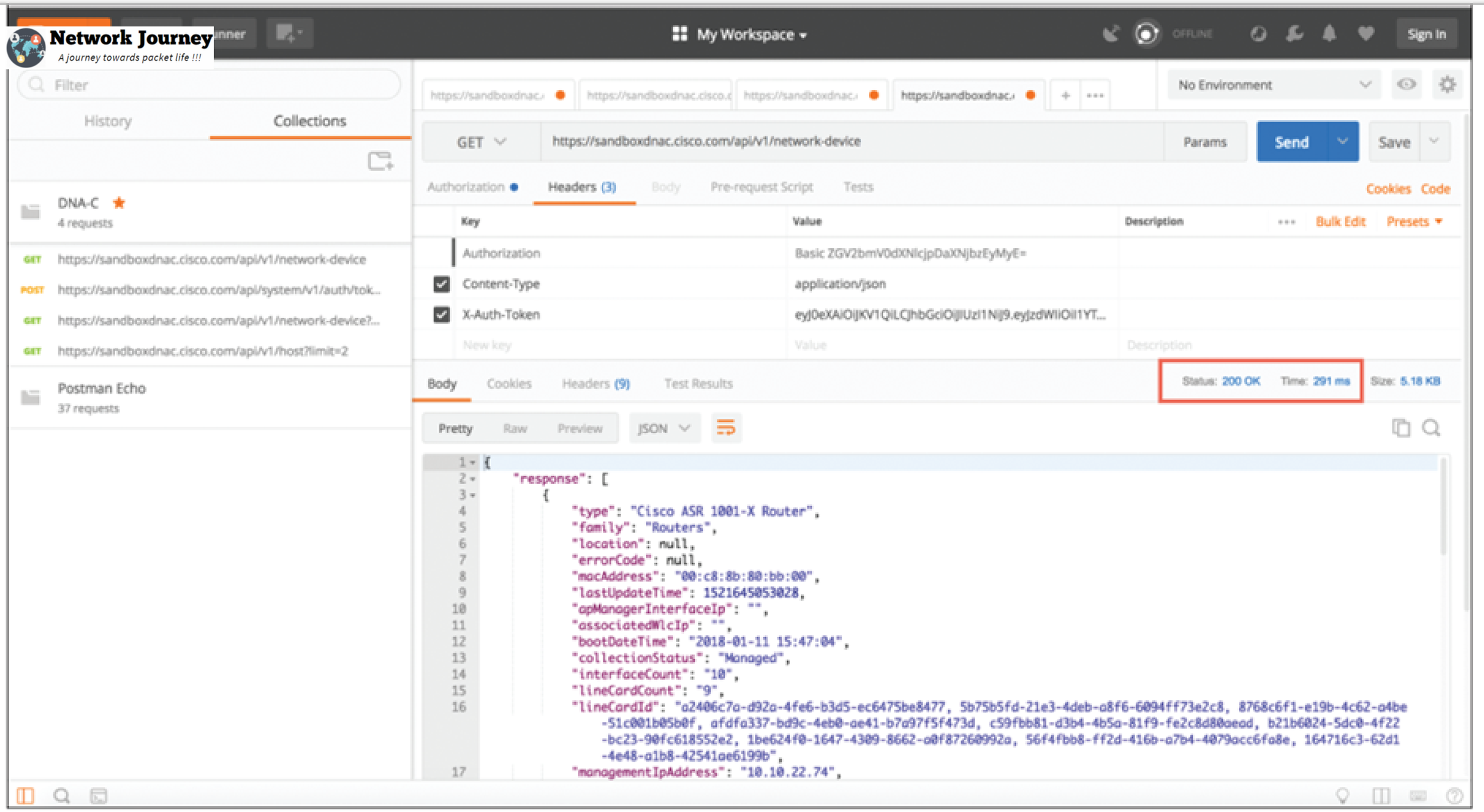


Figure 28-9 *Postman Setup for Retrieving the Network Device Inventory with an API Call*

Git is a distributed version control system

Git is a distributed version control system that allows developers to track changes in their source code, collaborate with other developers, and manage code repositories. Here is a list of some common Git operations:

1. git init: Initializes a new Git repository.
2. git clone: Creates a copy of a remote repository on your local machine.
3. git add: Adds changes in files to the staging area, in preparation for a commit.
4. git status: Shows the status of your working directory, including any changes made, staged, or uncommitted.
5. git diff: Displays the differences between your working directory and the most recent commit.
6. git commit: Records the changes in the staging area as a new commit in the repository, along with a descriptive message.
7. git log: Shows the commit history, including author, date, and commit message.
8. git branch: Lists all branches in your repository and indicates the current branch.
9. git checkout: Switches to a specified branch or commit.
10. git merge: Merges changes from one branch into another.
11. git pull: Fetches changes from a remote repository and merges them into your current branch.
12. git fetch: Retrieves updates from a remote repository but does not merge them.
13. git push: Sends your local commits to a remote repository.
14. git remote: Lists remote repositories connected to your local repository.
15. git remote add: Adds a new remote repository to your local repository.
16. git remote remove: Removes a remote repository from your local repository.
17. git stash: Temporarily saves changes in your working directory, allowing you to switch branches without committing.
18. git stash apply: Restores the changes saved with git stash.
19. git rebase: Reapplies commits on top of another base tip, often used to integrate changes from one branch into another.
20. git config: Allows you to set or update Git configuration options, like your username, email, and default editor.

Network Journey
A journey towards packet life !!!

There are many more Git commands and options, but these are some of the most commonly used ones. To learn more about Git, you can refer to the official documentation at Git is a distributed version control system.

Version Control

Version control, also known as source control or revision control, is the process of managing and tracking changes to source code, documents, or any other form of information over time. It enables multiple people to work on a project simultaneously and allows for easy retrieval of previous versions of the files. Version control systems provide several benefits, including:

1. Collaboration: Version control systems allow multiple team members to work on the same project without accidentally overwriting each other's changes. They facilitate efficient collaboration and communication among team members.
2. History and accountability: Version control systems keep a detailed log of all changes made to a project, including who made the change, when it was made, and why it was made. This helps in tracking progress and understanding how the project has evolved over time.
3. Reverting to previous versions: If a mistake is made or a bug is introduced, version control systems allow you to revert to a previous version of the project. This can be invaluable when trying to fix issues or recover from errors.
4. Branching and merging: Version control systems enable you to create branches, which are separate versions of a project that can be developed independently. This allows you to work on new features or bug fixes without affecting the main branch until the changes are ready to be merged.
5. Backup and recovery: Version control systems act as a backup, storing all versions of your project files. In case of data loss, you can recover your work from the version control system.

Network Journey
A journey towards packet life !!!

There are two main types of version control systems:

1. Centralized Version Control Systems (CVCS): These systems have a central server that stores all the versions of a project, and users must connect to the server to access the files. Examples of CVCS include Subversion (SVN) and Perforce.
2. Distributed Version Control Systems (DVCS): In these systems, each user has a complete copy of the repository, including the full history. This allows for better collaboration, offline work, and faster operations. Examples of DVCS include Git, Mercurial, and Bazaar.

Example: Git is one of the most popular version control systems, known for its flexibility, speed, and powerful branching and merging capabilities.

Git, GitHub, Gitlab

Git, GitHub, and GitLab are related but distinct tools that play crucial roles in version control, collaboration, and software development. Here's a brief explanation of each:

1. Git: Git is a distributed version control system created by Linus Torvalds, the creator of Linux. Git enables developers to track changes in their source code, collaborate with others, and manage code repositories. Git is a command-line tool that can be used locally on your computer or with remote repositories hosted on servers. Some of the key features of Git include branching, merging, and the ability to work offline with a complete copy of the repository.
2. GitHub: GitHub is a web-based platform that provides hosting for Git repositories and offers a user-friendly interface for managing and collaborating on projects. In addition to hosting Git repositories, GitHub provides features like issue tracking, pull requests, code review, and project management tools. GitHub fosters an open-source community where developers can contribute to public repositories or create their own private repositories. It's important to note that GitHub is not the only platform that offers Git repository hosting. Other platforms like GitLab and Bitbucket provide similar services.
3. GitLab: GitLab is an open-source platform for managing Git repositories, similar to GitHub. GitLab offers a comprehensive suite of tools for managing the software development lifecycle, including source code management, continuous integration and deployment (CI/CD), code review, issue tracking, and project management features. GitLab can be used as a cloud-hosted service (GitLab.com) or self-hosted on your own servers (GitLab Community Edition or GitLab Enterprise Edition). This self-hosting capability provides organizations with more control over their data and infrastructure, making GitLab a popular choice for businesses with strict security or compliance requirements.

In summary, Git is the version control system used to manage and track changes in source code, while GitHub and GitLab are web-based platforms that provide hosting and collaboration tools for Git repositories. While GitHub and GitLab offer similar features, they differ in terms of pricing, hosting options, and some additional features.

YANG

YANG (short for Yet Another Next Generation) is a data modeling language used to describe the structure and semantics of configuration data, state data, and notifications for network devices and services. YANG is human-readable, modular, and extensible, making it suitable for representing complex network configurations and operations.

YANG models are used in conjunction with network management protocols like NETCONF (Network Configuration Protocol) and RESTCONF (RESTful Configuration Protocol) to enable structured and programmatic configuration and management of network devices.

There are several reasons why YANG models are used in network management:

1. Standardized representation: YANG models provide a standardized way to represent the structure, constraints, and semantics of configuration data, state data, and notifications, ensuring consistency across devices and vendors.
2. Modularity and extensibility: YANG models are modular and can be extended using other YANG modules or submodules. This enables network operators to build complex configurations and services by combining multiple reusable YANG modules.
3. Tooling and code generation: YANG models can be used to generate code, documentation, and user interfaces, which can help automate and simplify network configuration and management tasks. Various tools exist for working with YANG models, such as pyang, yanglint, and YANG development kits (YDK) for different programming languages.
4. Vendor-neutral approach: YANG models enable a vendor-neutral approach to network management, which reduces vendor lock-in and allows network operators to use a single set of tools and processes to manage devices from different vendors.
5. Validation and error handling: YANG models define data constraints, such as data types, ranges, and patterns, which allow for validation of configuration data and error handling. This can help prevent configuration errors and improve the robustness of network management operations.
6. Improved automation and programmability: YANG models, combined with network management protocols like NETCONF and RESTCONF, provide a programmatic and structured way to configure and manage network devices. This enables better automation, orchestration, and integration with other management systems and tools.

In summary, YANG models provide a standardized, structured, and extensible way to represent configuration data, state data, and notifications for network devices and services. They are used in conjunction with network management protocols like NETCONF and RESTCONF to enable efficient, programmatic, and vendor-neutral network configuration and management.

What is data modeling language ?

A data modeling language is a formal language used to describe the structure, organization, constraints, and relationships of data in a specific domain. Data modeling languages are typically used in the design and development of databases, data processing systems, and data exchange formats. They allow developers, architects, and administrators to define and manipulate data structures in a consistent and standardized manner.

Data modeling languages can be categorized into two main types:

1. Textual data modeling languages: These languages use textual syntax to represent data structures and relationships. Examples of textual data modeling languages include:

- SQL DDL (Data Definition Language): Used to define and manage database structures like tables, columns, indexes, and constraints in relational databases.
- XML Schema: Used to define the structure, elements, and attributes of XML documents.
- JSON Schema: Used to describe the structure and validate JSON data.
- YANG (Yet Another Next Generation): A data modeling language used to define the structure and semantics of configuration data, state data, and notifications for network devices and services.

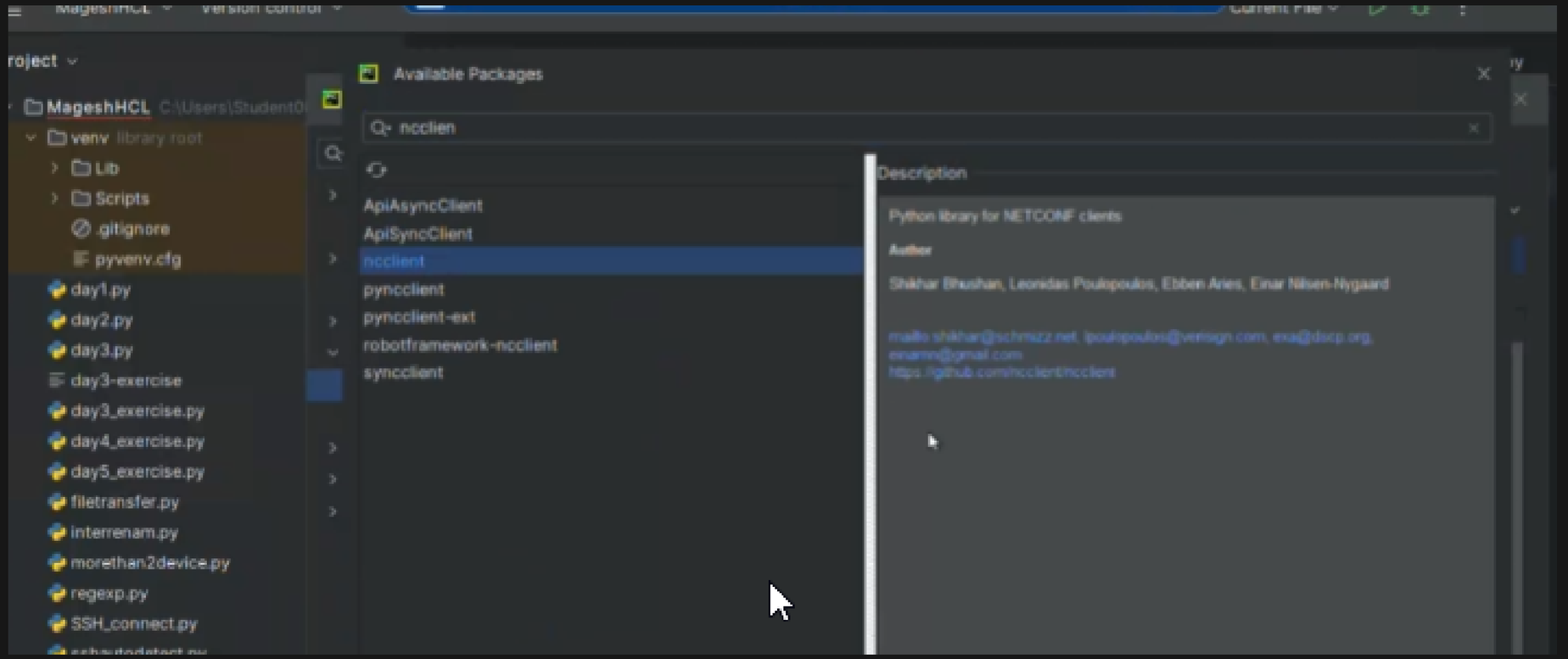
2. Graphical data modeling languages: These languages use visual notation to represent data structures and relationships. Examples of graphical data modeling languages include:

- Entity-Relationship (ER) diagrams: A graphical representation of entities, attributes, and relationships in a relational database.
- Unified Modeling Language (UML): A general-purpose modeling language used to represent the structure, behavior, and architecture of software systems. UML includes class diagrams for data modeling, which can represent entities, attributes, and relationships in a database.
- IDEF1X (Integration Definition for Information Modeling): A graphical notation used to represent the structure and relationships of data in relational databases.

Data modeling languages are essential for designing and implementing databases, data processing systems, and data exchange formats, as they provide a standardized and consistent way to describe and manage data structures and relationships.

NETCONF

```
CSR-XE-RTR#
CSR-XE-RTR#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
CSR-XE-RTR(config)#
CSR-XE-RTR(config)#
CSR-XE-RTR(config)#netc
CSR-XE-RTR(config)#netconf-ya
CSR-XE-RTR(config)#netconf-yang
CSR-XE-RTR(config)#
```


MageshHCL
venv library root
Lib
Scripts
.gitignore
pyvenv.cfg
day1.py
day2.py
day3.py
day3-exercise
day3_exercise.py
day4_exercise.py
day5_exercise.py
filetransfer.py
interrenam.py
morethan2device.py
regexp.py
SSH_connect.py
Available Packages
ncclien
ApiAsyncClient
ApiSyncClient
ncclient
pyncclient
pyncclient-ext
robotframework-ncclient
syncclient
Description
Python library for NETCONF clients
Author
Shikhar Bhushan, Leonidas Poulopoulos, Ebben Aries, Einar Nilsen-Nygaard

SCRIPT#1


```
from ncclient import manager

device_ip = "192.168.1.1"
username = "your_username"
password = "your_password"

with manager.connect(host=device_ip, port=830, username=username, password=password, hostkey_verify=False) as m:
    running_config = m.get_config(source="running")
    print(running_config.xml)
```

SCRIPT#2


```
from ncclient import manager

device_ip = "192.168.1.1"
username = "your_username"
password = "your_password"

interfaces_filter = '''
<filter>
   <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
      <interface/>
   </interfaces>
</filter>
'''

with manager.connect(host=device_ip, port=830, username=username, password=password, hostkey_verify=False) as m:
   response = m.get_config(source='running', filter=interfaces_filter)
   print(response.xml)
```

SCRIPT#3

Network Journey
A journey towards packet life !!!

```
from ncclient import manager
deviceip ="172.16.24.184"
user123 = "admin"
pass123 = "cisco"

loopback_config_payload = '''
<config>
  <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
  <name>Loopback555</name>
  <description>student500 - testing HCL</description>
  <enabled>true</enabled>
  <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift:softwareLoopback</type>
    </interface>
   </interfaces>
</config>
'''

with manager.connect(host=deviceip, port=830,username=user123,password=pass123,hostkey_verify=False) as m123:
    running_config123 = m123.edit_config(target="running", config=loopback_config_payload)
    print(running_config123.xml)
```

SCRIPT#4


```
from ncclient import manager

device_ip = "192.168.1.1"
username = "your_username"
password = "your_password"

interface_ip_config = '''
<config>
   <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
      <interface>
         <name>GigabitEthernet1</name>
         <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
            <address>
               <ip>192.168.2.1</ip>
               <netmask>255.255.255.0</netmask>
            </address>
         </ipv4>
      </interface>
   </interfaces>
</config>
'''

with manager.connect(host=device_ip, port=830, username=username, password=password, hostkey_verify=False) as m:
   m.edit_config(target='running', config=interface_ip_config)
```

SCRIPT#5


```
from ncclient import manager

device_ip = "192.168.1.1"
username = "your_username"
password = "your_password"

delete_loopback_config = '''
<config>
   <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
      <interface operation="delete">
         <name>Loopback100</name>
      </interface>
   </interfaces>
</config>
'''

with manager.connect(host=device_ip, port=830, username=username, password=password, hostkey_verify=False) as m:
   m.edit_config(target='running', config=delete_loopback_config)
```

PYATS (Python Automation Test System)

Installation of pyATS Environment (Full, Genie, Xpresso)

To install the pyATS environment, which includes Full, Genie, and Xpresso, follow these steps:

1. Ensure you have Python 3.6 or later installed. You can check your Python version with the following command:

python --version
or
python3 --version

2. If you don't have Python installed or need to upgrade, download the appropriate version from the official Python website: https://www.python.org/downloads/

3. Create a virtual environment (optional but recommended) to isolate the pyATS installation from other Python projects. To create a virtual environment, run the following command:

sudo apt install python3.10-venv *(press Y for "yes" to download all dependencies)*
python3 -m venv pyats-env

Replace pyats-env with the desired name for your virtual environment.

4. Activate the virtual environment. The activation command depends on your operating system:

- For macOS and Linux:

```
source pyats-env/bin/activate
```

5. Update pip (Python package manager) to the latest version:

```
pip install --upgrade pip
```

or

```
apt install python3-pip
```

6. Install the pyATS package, including the Genie and Xpresso libraries:

```
pip install pyats[full]
```

This command installs the complete pyATS package, which includes Genie and Xpresso.

7. Verify the installation by running the following command:

```
pyats version check
```

This command displays the versions of pyATS, Genie, and their dependencies. If the installation was successful, you should see the versions listed without any errors.

8. To deactivate the virtual environment when you're done working with pyATS, run the following command:

```
deactivate
```

Now you have the pyATS environment with Full, Genie, and Xpresso installed and ready to use.

Network Journey
A journey towards packet life !!!

Network Journey
A journey towards packet life !!!

PYATS

COMMANDS FOR ACTIVATING PYATS BEFORE YOU START WORKING ON PYATS ENVIRONMENT

1. For macOS and Linux:

```
source pyats-env/bin/activate
```

2. Verify the installation by running the following command:

```
pyats version check
```

1. Basic Test

```
sudo apt install git

# clone example folder
sudo apt install git
git clone https://github.com/CiscoTestAutomation/examples
cd examples

# start with executing the basic examples jobfiles.
# this is a basic example demonstrating the usage of a jobfiles,
# and running through a single aetest testscript.

pyats run job basic/basic_example_job.py
```

Pro Tip

Try the following command to view your logs:

```
pyats logs view
```

CREATE TESTBED

(device inventory)

Parser view website
https://pubhub.devnetcloud.com/media/genie-feature-browser/docs/#/parsers

Create TestBed Device YAML

METHOD#1 Manual Method using Json2Yaml.com:

You can enter the device data manually, without having to first create a YAML or Excel/CSV file:

METHOD#2 Create a Testbed from an Excel sheet

You can also create a testbed yaml from an excel sheet with pyats create testbed command, here's an example:

Here is an excel sheet containing device data:
reference : https://pubhub.devnetcloud.com/media/genie-docs/docs/cookbooks/genie.html#create-a-testbed-from-a-dictionary
download sample excel : https://pubhub.devnetcloud.com/media/pyats-getting-started/docs/quickstart/manageconnections.html

Sample Excel file
we can turn it into a testbed yaml file by running the following command:

[genie] demo:373> pyats create testbed file --path my_devices.xls --output yaml/my_testbed.yaml

Validate testbed
pyats validate testbed examples/yaml/my_testbed.yaml

METHOD#3 Interactive Method
pyats create testbed interactive --output yaml/my_testbed.yaml --encode-password

-refer next slide

Network Journey
A journey towards packet life !!!

```
(pyats-env) student24@student24-virtual-machine:~/examples$ pyats create testbed
 interactive --output yaml/my_testbed.yaml --encode-password
Start creating Testbed yaml file ...
Do all of the devices have the same username? [y/n] y
Common Username: admin

Do all of the devices have the same default password? [y/n] y
Common Default Password (leave blank if you want to enter on demand): cisco

Do all of the devices have the same enable password? [y/n] y
Common Enable Password (leave blank if you want to enter on demand): cisco123


Device hostname: CSR1
    IP (ip, or ip:port): 172.16.24.59
    Protocol (ssh, telnet, ...): ssh
    OS (iosxr, iosxe, ios, nxos, linux, ...): iosxe
More devices to add ? [y/n] n
```

FIX SSH CONNECTIVITY FROM UBUNTU TO ROUTERS

Network Journey
A journey towards packet life !!!

FIX:

goto ubuntu terminal and copy paste two lines given below in white font:

sudo su

nano /etc/ssh/ssh_config

now text editor is opened, copy paste 3 to 4 lines of code given below in white font

contrl + x

y for yes

enter to save

Ciphers aes256-cbc,aes128-ctr,aes192-ctr,aes256-ctr,aes128-cbc,3des-cbc

KexAlgorithms +diffie-hellman-group1-sha1,diffie-hellman-group-exchange-sha1,diffie-hellman-group14-sha1

HostkeyAlgorithms +ssh-rsa,ssh-dss

EXECUTION

Connect to Device

https://pubhub.devnetcloud.com/media/pyats-getting-started/docs/quickstart/parseoutput.html

METHOD#1: via CLI COMMAND

Inititate the CLI for fetch "show version"

(pyats-env) networkjourney@networkjourney-virtual-machine:~/pyats-env$ pyats parse "show version" --testbed-file yaml/my_testbed.yaml --devices SW01 SW02 SW03

METHOD#2: via Python IDLE mode

get inside python idle by typing below command>

python

and then copy paste below code in python idle


```
from genie.testbed import load
tb = load('yaml/my_testbed.yaml')
dev = tb.devices['SW01']
dev.connect()

p1 = dev.parse('show inventory')
print(p1)
print('My serial for slot1 is: ' + p1['main']['chassis']['CSR1000v']['sn']) #dictionary key change per ios family, so first fetch print(p1) and accondingly update on access dictionary key

p2 = dev.parse("show clock")
print("MY SWITCH CLOCK IS: " + str(p1))
print("MY SW01 DATE IS: " + str(p1["day"]))  #dictionary key change per ios family, so first fetch print(p1) and accondingly update on access dictionary key

access from dictionary:
print(p1)
print(p2)
p2["dictionary-element"]["key"]

control + D (to exist from python IDLE)
```

METHOD3: Integrate with pycharm

Goto > ubuntu terminal

sudo snap install pycharm-community --classic --channel=2022.3

note: to remove use this command sudo snap remove pycharm-community. To open pycharm > left bottom dotted button > search for "pycharm"

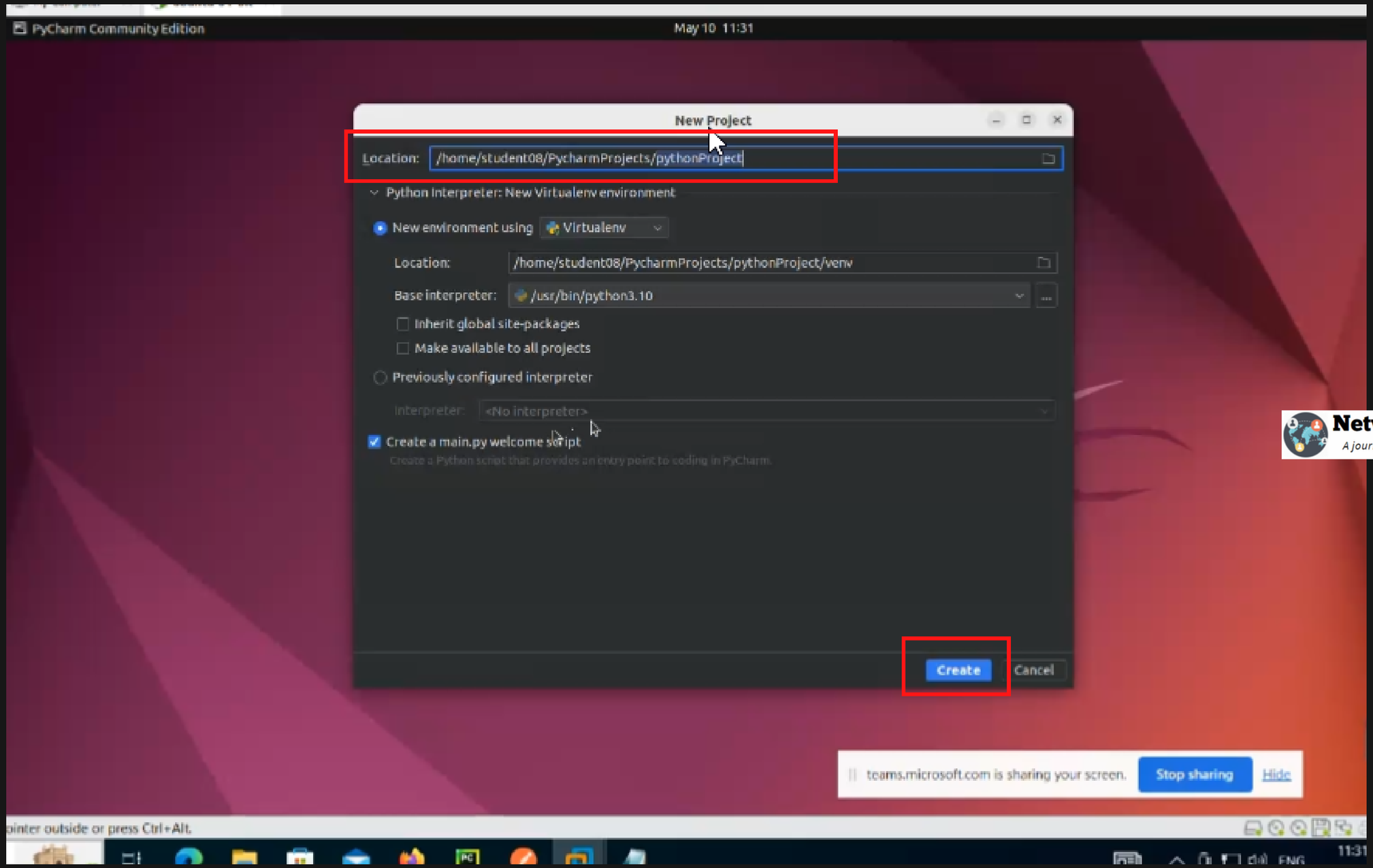
Goto pycharm > open Pyats-env folder

pycharm > open Pyats-env folder > Project Interpreter > Existing Environment > /home/networkjourney/pyats-env/bin/python3.10 *(all libraries will pre-load from pyats-env)*

create new example.py in pycharm and paste below script and execute:

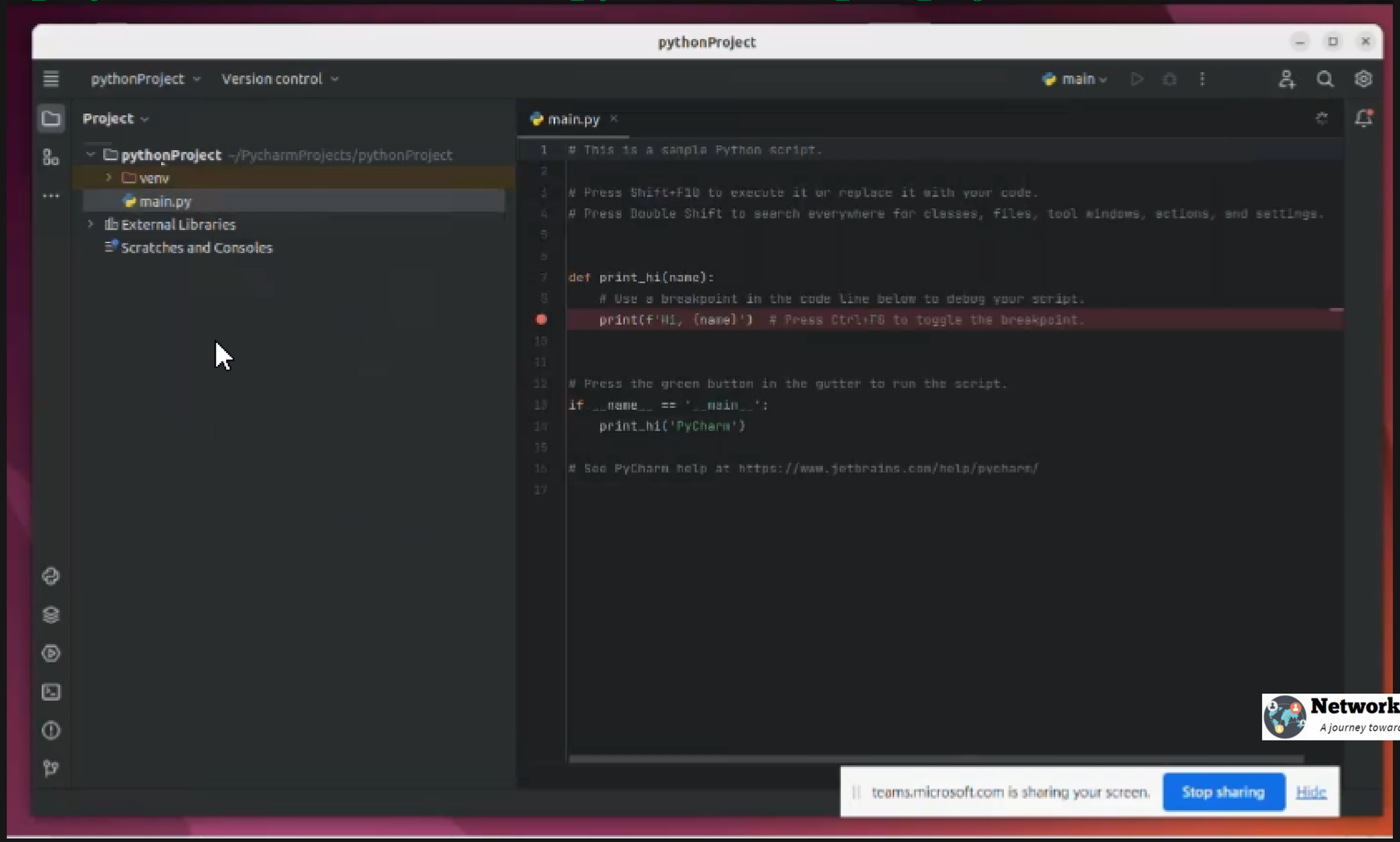
```
from genie.testbed import load
tb = load('yaml/my_testbed.yaml')
dev = tb.devices['SW01']
dev.connect()
p1 = dev.parse('show ip interface  brief')
print(p1)
```

Step1: once pycharm downloaded from method#3 follow these screenshots



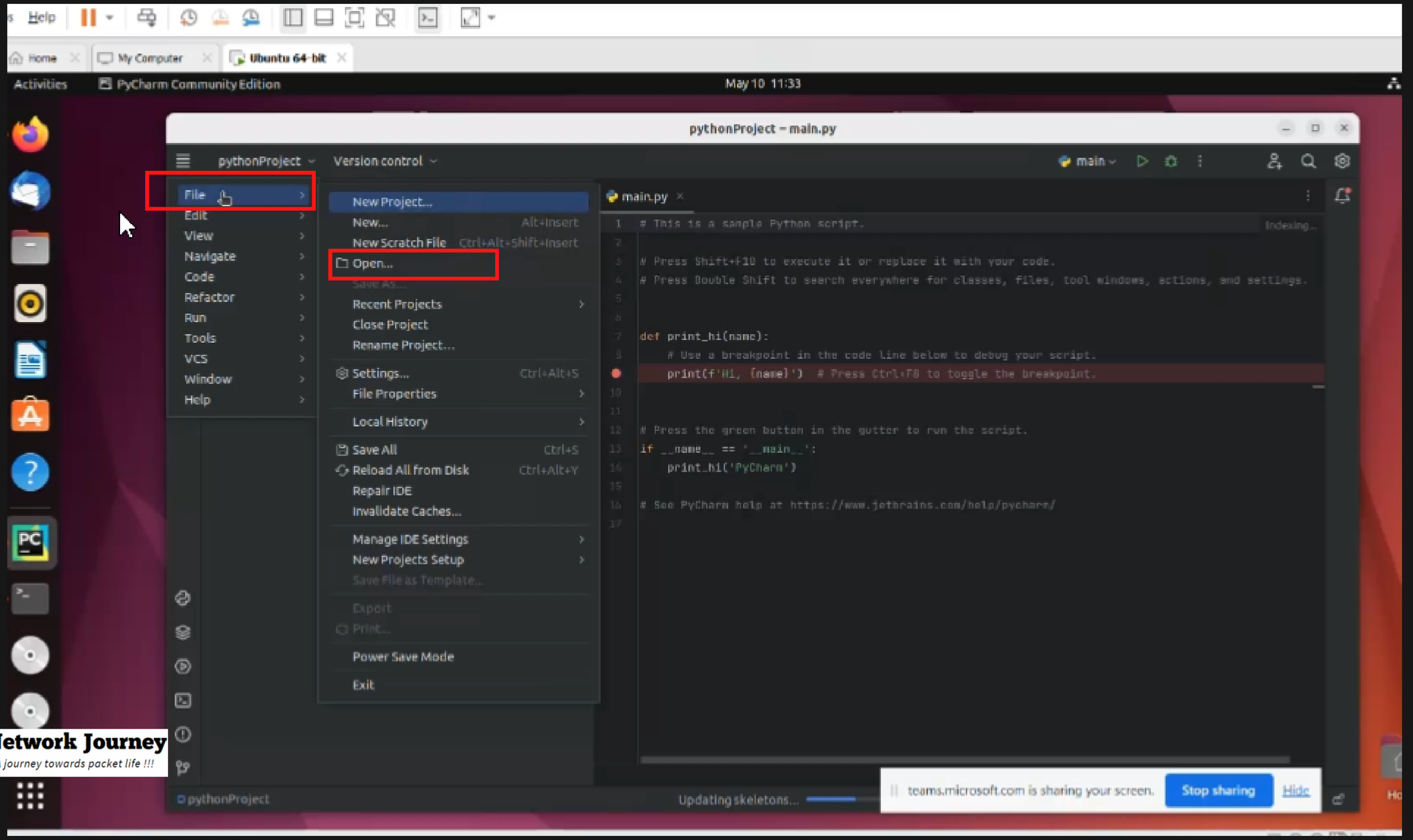
Network Journey
A journey towards packet life !!!

step2: you land here and in next step you click on "open" projects



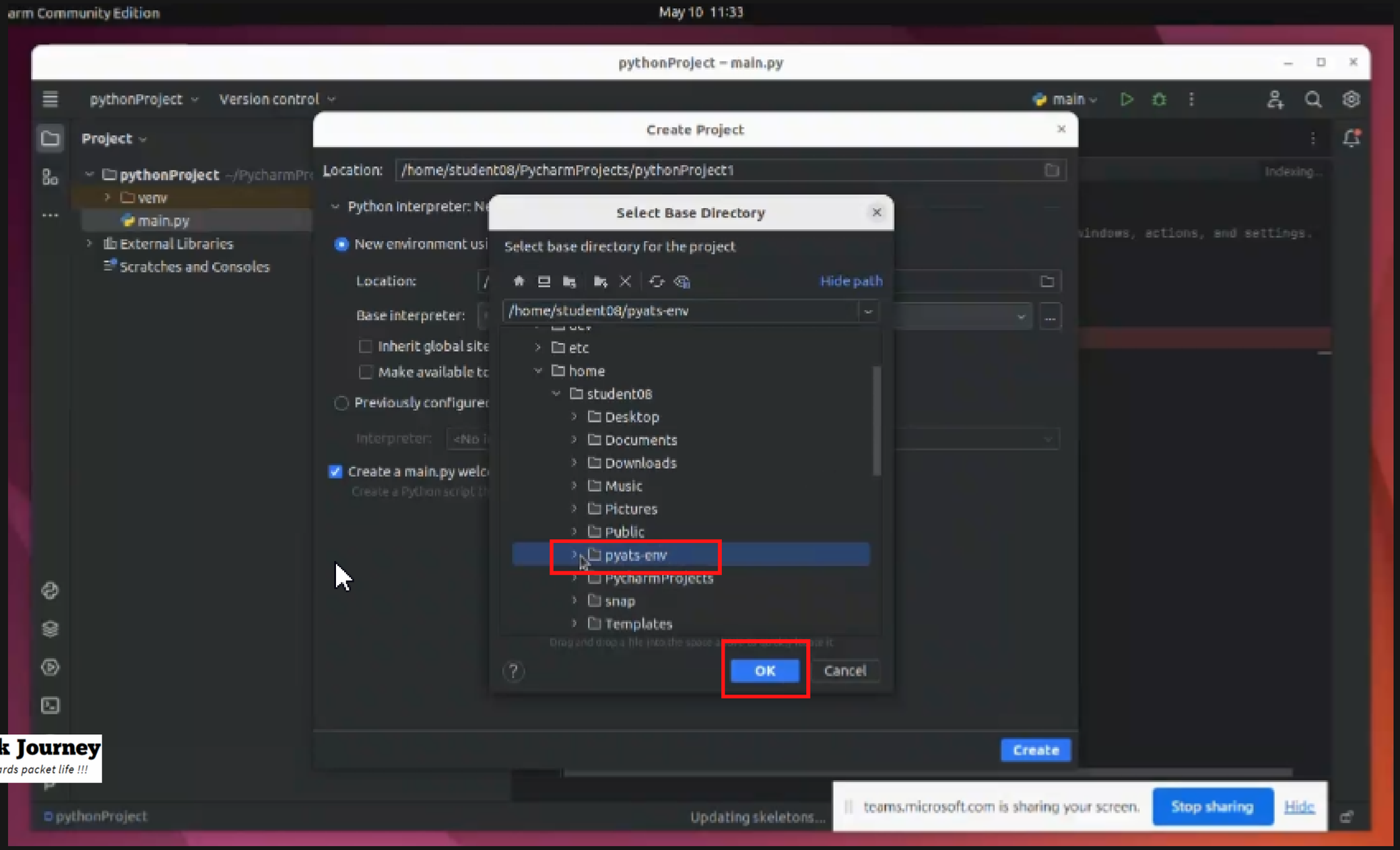
Network Journey
A journey towards packet life !!!

step3: keep pycharmprojects and pyats-env into two seperate environments



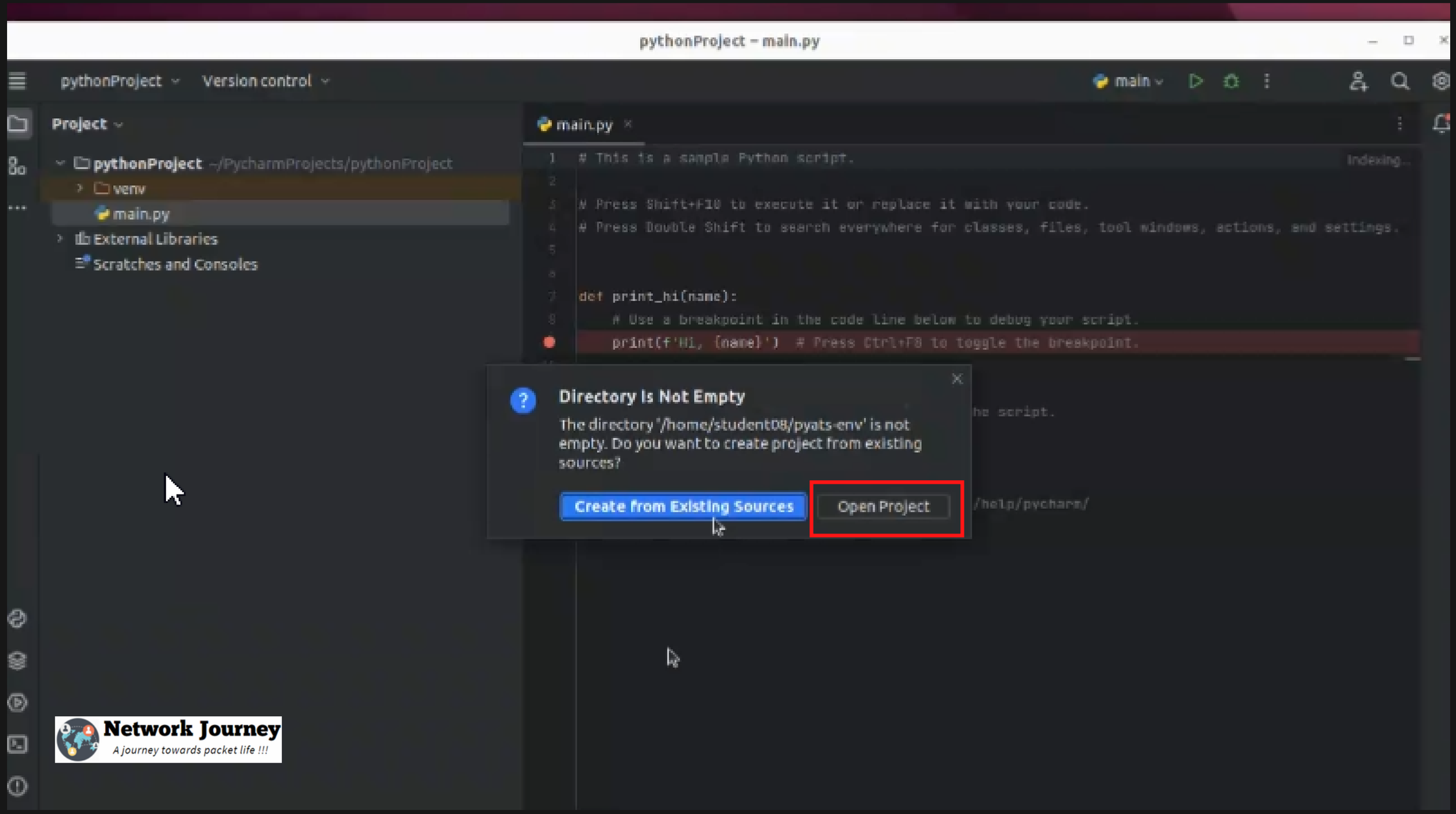
Network Journey
A journey towards packet life !!!

step4: select "pyats-env" folder

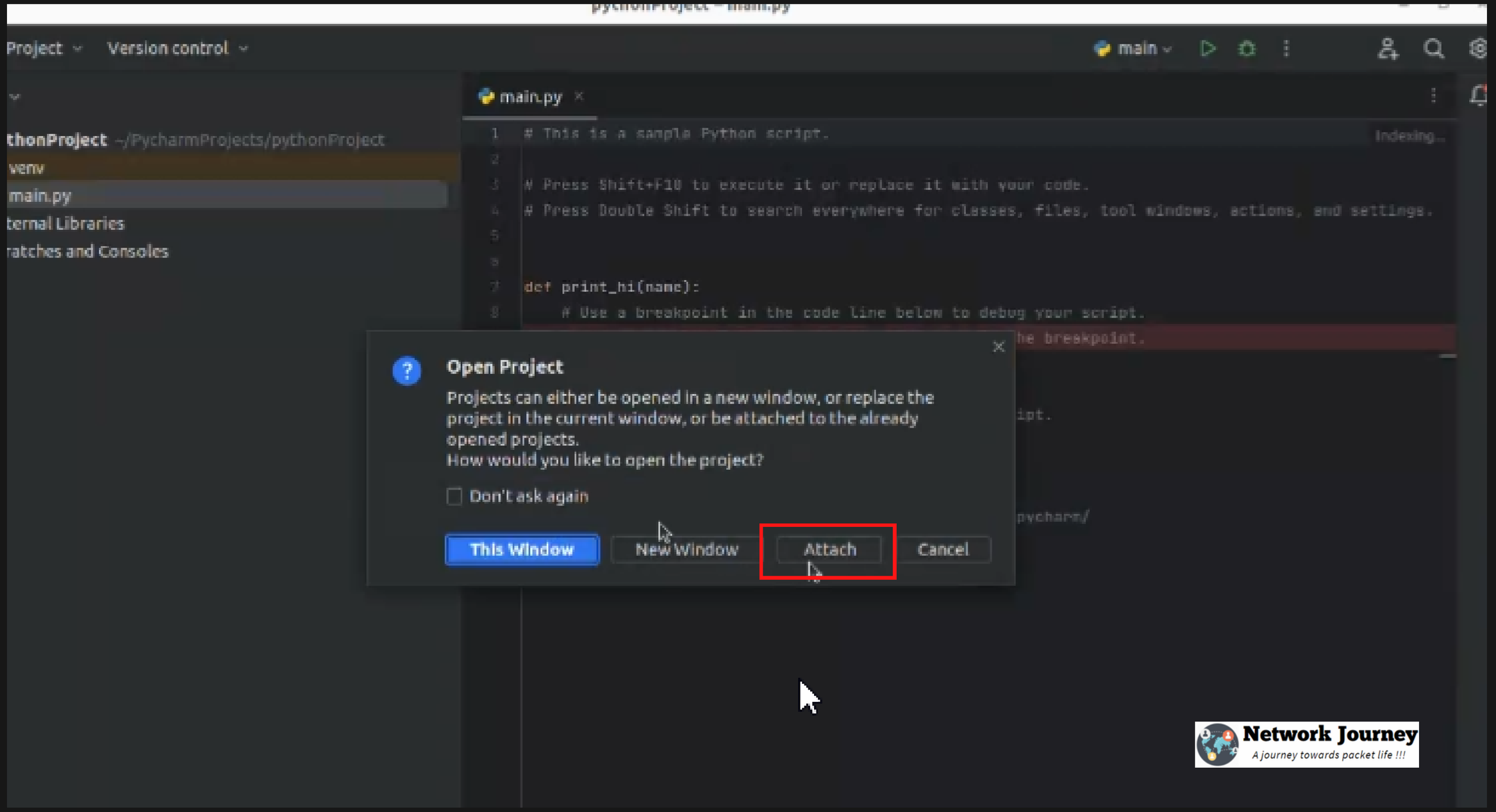


Network Journey
A journey towards packet life !!!

step5: (optional)

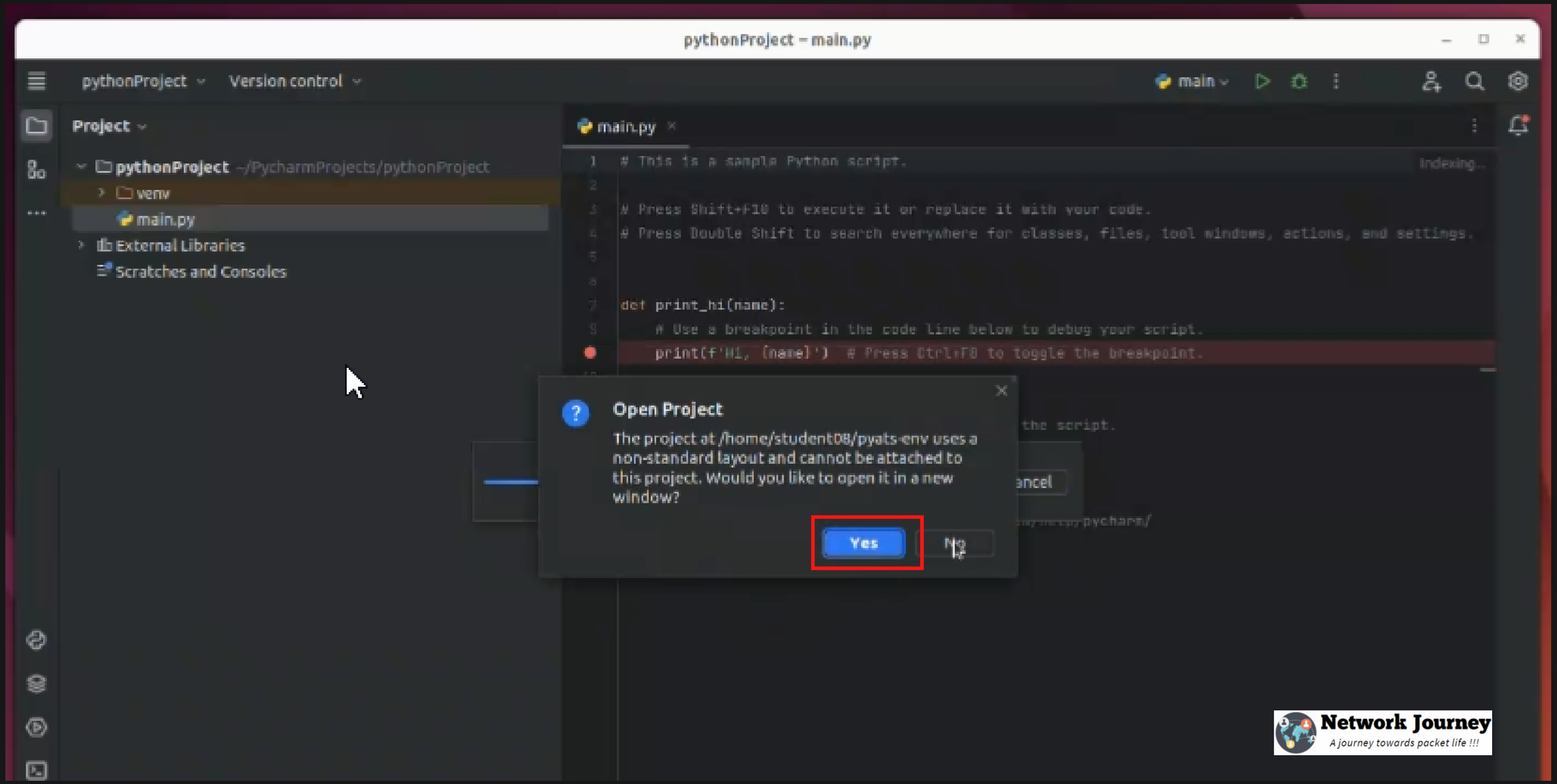


step6: click on "attach"

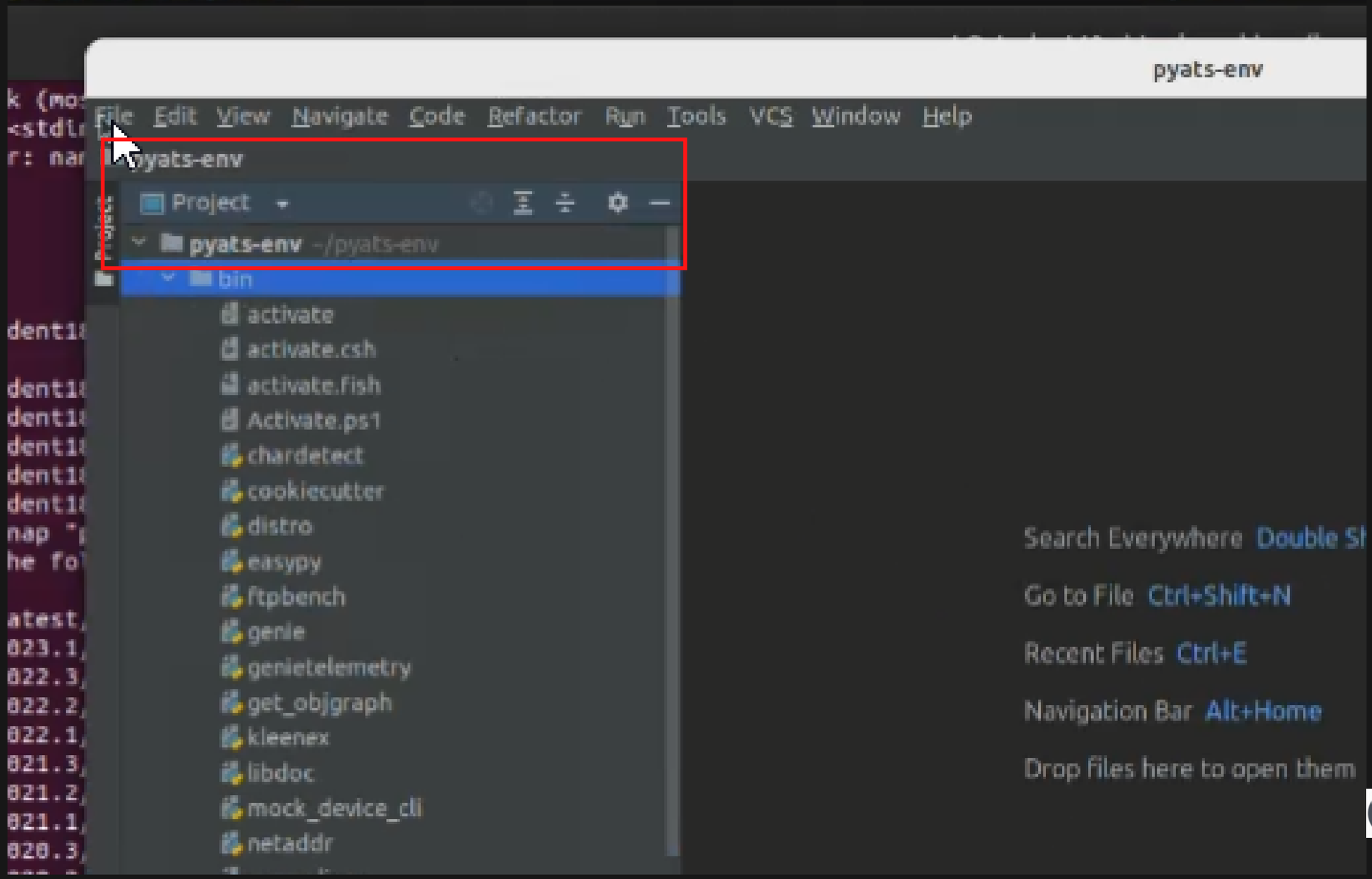


Network Journey
A journey towards packet life !!!

step7: click on "no" (optional)

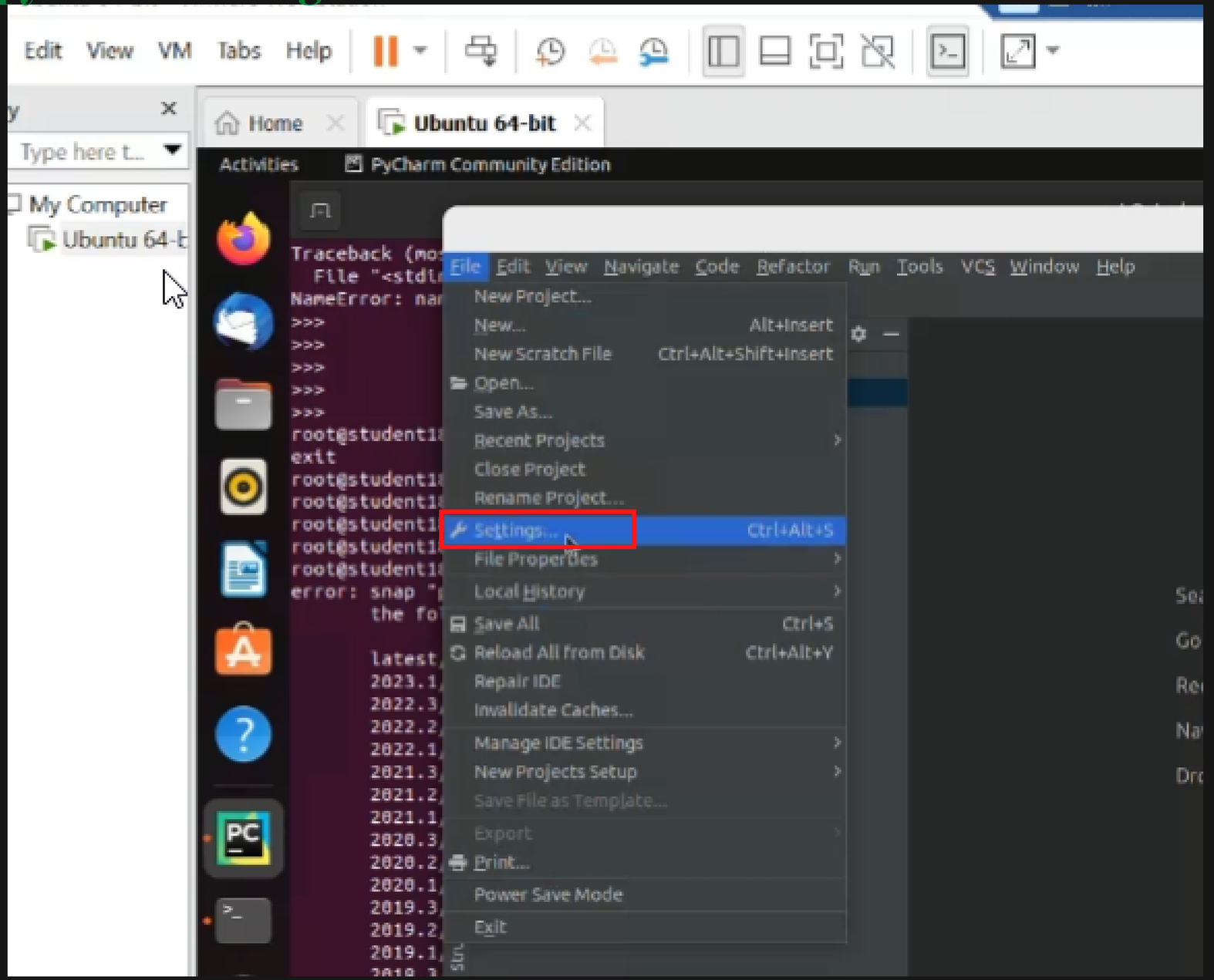


step8: You will see "pyats-env" project on your pycharm



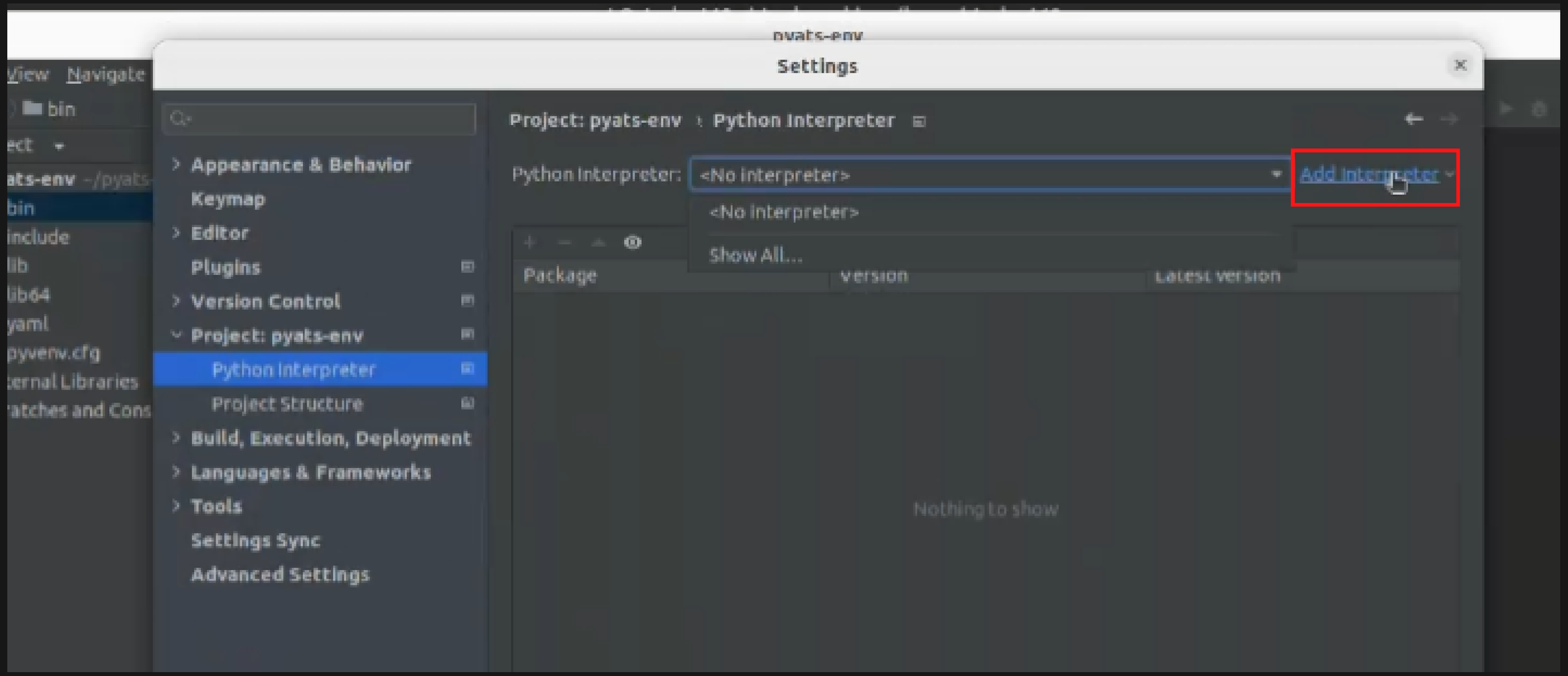
Network Journey
A journey towards packet life !!!

step9: goto pycharm settings



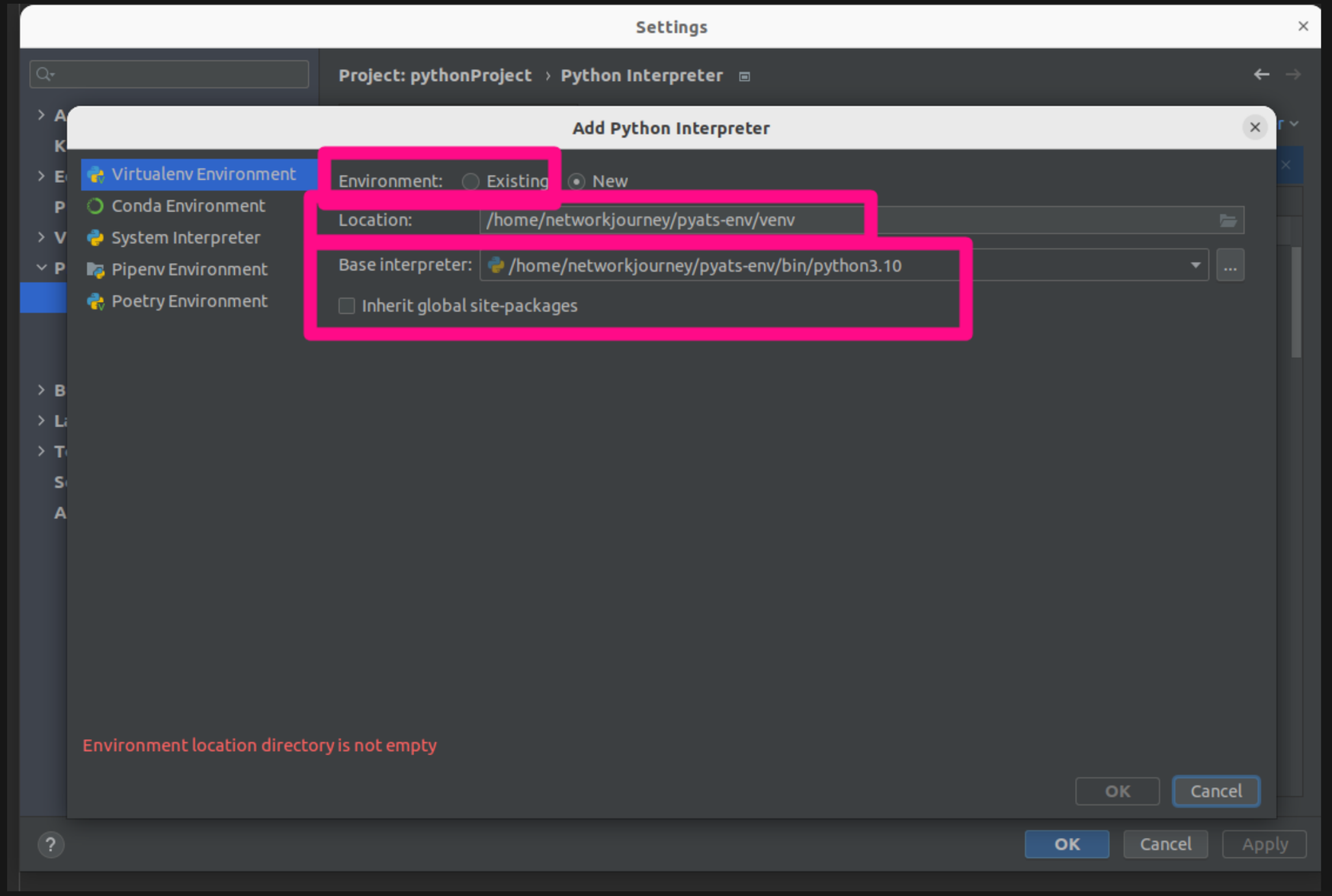
Network Journey
A journey towards packet life !!!

step10: goto "ADD INTERPRETER"



Network Journey
A journey towards packet life !!!

step11: select "EXISTING" > modify "BASIC INTERPRETER"



Network Journey
A journey towards packet life !!!

FIX:

If pycharm not letting you create new file, it could be due to file permission.

To change Folder root permission in ubuntu operating systems:

```
sudo su
chmod ugo+rwx <folder-name>
```

Network Journey
A journey towards packet life !!!

Perform Network Validation Tests for Networking Protocols like Static, OSPF, BGP using Genie

1. BGP VALIDATION:

This script connects to the device, learns the BGP state, and performs two validation tests:

Run below Tests:

Check if the BGP session state is "established" for each neighbor.

Check if the BGP session uptime is greater than 60 seconds for each neighbor.

You can add more tests as needed to validate your BGP configuration.

Network Journey
A journey towards packet life !!!

```
from pyats.topology import loader
from genie.libs.ops.bgp.ios.bgp import Bgp

# Load testbed file
testbed = loader.load('testbed.yml')
device = testbed.devices['SW01']

# Connect to the device
device.connect()

# Create a BGP object and learn BGP state
bgp = Bgp(device=device)
bgp.learn()
print("BGP Info:", bgp.info)  # Debug print


# Access the default VRF BGP neighbors
bgp_vrf = bgp.info.get('instance', {}).get('default', {}).get('vrf', {}).get('default', {})
print("BGP VRF Data:", bgp_vrf)  # Debug print

bgp_neighbors = bgp_vrf.get('neighbor', {})
print("BGP Neighbors:", bgp_neighbors)  # Debug print

# Perform validation tests
for neighbor, neighbor_data in bgp_neighbors.items():
    print(f"Checking BGP neighbor: {neighbor}")
    print(f"Neighbor data: {neighbor_data}")  # Debug print

    # Test: BGP session state should be "established"
    is_established = neighbor_data['session_state'] == 'Established'
    print(f"  BGP session established: {is_established}")
    assert is_established, f"BGP session not established for neighbor {neighbor}"

    # Test: Check if 'session_uptime' key is present and the value is greater than 60
    if 'session_uptime' in neighbor_data and neighbor_data['session_uptime'] > 60:
        print(f"  BGP session uptime greater than 60 seconds: True")
    else:
        print(f"  BGP session uptime greater than 60 seconds: False")
        # Handle the case when session uptime is less than 60 seconds
        # Add your desired behavior or raise an exception if required

    print(f"  All tests passed for neighbor {neighbor}\n")

    # Add more tests as needed
```

Network Journey
A journey towards packet life !!!

Output found:

The updated code is working as expected. It correctly identifies that the BGP session is not established for the neighbor with IP address 192.168.61.136 and raises an assertion error. The code also handles the case when the 'session_uptime' key is not present, so it doesn't cause a KeyError.

Here's the summary of the output:

For neighbor 192.168.61.138, the BGP session is established and the session uptime is less than 60 seconds. The code prints that all tests passed for this neighbor.
For neighbor 192.168.61.136, the BGP session is not established. The code raises an assertion error, indicating that the BGP session is not established for this neighbor.
If you have any further questions or need additional assistance, feel free to ask.

The script will perform the BGP validation tests on the devices specified in the testbed file. If a test fails, you will see an AssertionError message with a [illegible] of the failed test. If all tests pass, there will be no output.

Network Journey
A journey towards packet life !!!

2. OSPF VALIDATION:

configure OSPF

R1:
router ospf 100
network 172.16.24.0 0.0.0.255 area 0

R2:
router ospf 100
network 172.16.24.0 0.0.0.255 area 0

Network Journey
A journey towards packet life !!!

```
from pyats.topology import loader
from genie.libs.ops.ospf.ios.ospf import Ospf
from genie.libs.parser.utils import get_parser

# Load testbed file
testbed = loader.load('testbed.yml')
mydevices = ["SW01", "SW02"]
for x in mydevices:
    device = testbed.devices[x]

    # Connect to the device
    device.connect()

    # Create an OSPF object and learn OSPF state
    ospf = Ospf(device)
    ospf.learn()

    # Access the default VRF OSPF areas
    ospf_vrf = ospf.info['vrf']['default']
    ospf_instances = ospf_vrf.get('address_family', {}).get('ipv4', {}).get('instance', {})

    # Perform validation tests
    for instance_id, instance_data in ospf_instances.items():
        ospf_areas = instance_data.get('areas', {})
        for area, area_data in ospf_areas.items():
            for intf, data in area_data['interfaces'].items():
                print(f"Interface: {intf}, Data: {data}")
                # Check if the interface is a sham-link
                if intf.startswith('SL'):
                    continue  # Skip validation tests for sham-link interfaces

                if not data.get('enable'):
                    print(f"OSPF not enabled on {intf}")
                else:
                    assert area == '0.0.0.0', f"Interface {intf} not in Area 0"
                    assert data['state'] == 'point-to-point', f"Interface {intf} not in point-to-point state"

    # Perform other OSPF-related tests, such as verifying neighbors, routes, and more
```

Network Journey
A journey towards packet life !!!

Validation:

```
SW01#
Interface: GigabitEthernet0/0, Data: {'name': 'GigabitEthernet0/0', 'interface_type': 'broadcast', 'passive': False, 'demand_circuit': False, 'priority': 1,
'transmit_delay': 1, 'bfd': {'enable': False}, 'hello_interval': 10, 'dead_interval': 40, 'retransmit_interval': 5, 'lls': True, 'enable': True, 'cost': 1, 'state': 'bdr',
'hello_timer': '00:00:04', 'dr_router_id': '192.168.61.140', 'dr_ip_addr': '192.168.61.140', 'bdr_router_id': '192.168.61.139', 'bdr_ip_addr': '192.168.61.139',
'neighbors': {'192.168.61.140': {'neighbor_router_id': '192.168.61.140', 'address': '192.168.61.140', 'dr_ip_addr': '192.168.61.140', 'bdr_ip_addr': '192.168.61.139',
'state': 'full', 'dead_timer': '00:00:35', 'statistics': {'nbr_event_count': 6, 'nbr_retrans_qlen': 0}}}}
Traceback (most recent call last):
  File "/home/networkjourney/pyats-env/genieconfscript.py", line 36, in <module>
    assert data['state'] == 'point-to-point', f"Interface {intf} not in point-to-point state"
AssertionError: Interface GigabitEthernet0/0 not in point-to-point state

Process finished with exit code 1
```

You must receive this error to see the "assertion error" printed

Network Journey
A journey towards packet life !!!

3. STATIC ROUTE:

configure static route

R1:
ip route 1.1.1.1 255.255.255.0 gi0/0

Network Journey
A journey towards packet life !!!

```
from genie.testbed import load

# Load the testbed file
testbed = load('testbed.yml')

# Connect to the device
device = testbed.devices['SW01']
device.connect()

# Execute the show command to retrieve the static route information
output = device.execute('show ip route static')

# Parse the output using the Genie parse method
parsed_output = device.parse('show ip route static')
print(parsed_output)
# Access the static_routes dictionary from parsed_output
static_routes = parsed_output['vrf']['default']['address_family']['ipv4']['routes']

for route, route_info in static_routes.items():
    if route_info['source_protocol'] == 'static':
        if 'next_hop_list' in route_info['next_hop']:
            for index, next_hop_info in route_info['next_hop']['next_hop_list'].items():
                #Test: Ensure the administrative distance is correct
                assert route_info['route_preference'] == 1, f"Incorrect administrative distance for route {route}"

                #Test: Ensure the next-hop IP address is valid
                assert '0.0.0.0' not in next_hop_info['next_hop'], f"Invalid next-hop IP address for route {route}"

        # Test: Ensure the outgoing interface is valid, if present
        if 'outgoing_interface' in route_info['next_hop']:
            for interface_name, interface_info in route_info['next_hop']['outgoing_interface'].items():
                assert interface_info['outgoing_interface'] != '', f"Invalid outgoing interface for route {route}"

        print(f"All tests passed for route {route}")

# Disconnect from the device
device.disconnect()
```

Network Journey
A journey towards packet life !!!

You must the below ASSERTIONERROR

```
SW01#
{'vrf': {'default': {'address_family': {'ipv4': {'routes': {'0.0.0.0/0': {'route': '0.0.0.0/0', 'active': True, 'metric': 0, 'route_preference': 254,
'source_protocol_codes': 'S*', 'source_protocol': 'static', 'next_hop': {'next_hop_list': {1: {'index': 1, 'next_hop': '192.168.61.2'}}}}}, '1.1.1.1/32':
{'route': '1.1.1.1/32', 'active': True, 'source_protocol_codes': 'S', 'source_protocol': 'static', 'next_hop': {'outgoing_interface':
{'GigabitEthernet0/0': {'outgoing_interface': 'GigabitEthernet0/0'}}}}}}}}}}
Traceback (most recent call last):
  File "/home/networkjourney/pyats-env/example1.py", line 24, in <module>
    assert route_info['route_preference'] == 1, f"Incorrect administrative distance for route {route}"
AssertionError: Incorrect administrative distance for route 0.0.0.0/0
```

Network Journey
A journey towards packet life !!!

You must the below ASSERTIONERROR

```
SW01#
{'vrf': {'default': {'address_family': {'ipv4': {'routes': {'0.0.0.0/0': {'route': '0.0.0.0/0', 'active': True, 'metric': 0, 'route_preference': 254,
'source_protocol_codes': 'S*', 'source_protocol': 'static', 'next_hop': {'next_hop_list': {1: {'index': 1, 'next_hop': '192.168.61.2'}}}}}, '1.1.1.1/32':
{'route': '1.1.1.1/32', 'active': True, 'source_protocol_codes': 'S', 'source_protocol': 'static', 'next_hop': {'outgoing_interface':
{'GigabitEthernet0/0': {'outgoing_interface': 'GigabitEthernet0/0'}}}}}}}}}}
Traceback (most recent call last):
  File "/home/networkjourney/pyats-env/example1.py", line 24, in <module>
    assert route_info['route_preference'] == 1, f"Incorrect administrative distance for route {route}"
AssertionError: Incorrect administrative distance for route 0.0.0.0/0
```

Network Journey
A journey towards packet life !!!

EASYPY

easypy is a command-line utility provided by the PyATS framework. It simplifies the execution of test scripts by providing a standardized way to run tests and generate test reports. Here are some reasons why easypy is commonly used:

1. Test Execution: easypy allows you to easily execute test scripts without the need for complex command-line arguments or configurations. It automatically handles the setup, execution, and cleanup of your tests.
2. Testbed Management: easypy provides built-in support for managing testbeds. You can define your network topology and device connections in a testbed file, which easypy can load and use during test execution.
3. Reporting and Results: easypy generates comprehensive test reports that provide detailed information about the test execution, including the test results, logs, and any errors or exceptions encountered. These reports make it easier to analyze the test results and identify any issues.
4. Integration with Automation Frameworks: easypy seamlessly integrates with various automation frameworks, such as PyATS and Genie. It allows you to leverage the capabilities of these frameworks to create robust and scalable automation solutions.
5. Parallel Execution: easypy supports parallel execution of test scripts, allowing you to run multiple tests simultaneously. This can significantly reduce the overall test execution time and increase efficiency.

Overall, easypy simplifies the test execution process, enhances test management, and provides useful reporting features, making it a valuable tool for test automation and network validation.

Network Journey
A journey towards packet life !!!

easypy can also be used as a job aggregator. In addition to running individual test scripts, easypy can aggregate multiple test scripts into a single job, allowing you to execute them as a group.
With easypy as a job aggregator, you can define a job file that specifies the test scripts to include in the job, along with any specific configurations or parameters for each test script. easypy will then execute each test script sequentially or in parallel, depending on the job configuration.
The job file can include multiple test scripts, each with its own set of tests and testcases. easypy will aggregate the results from each test script and generate a consolidated test report that provides an overview of the entire job execution.

Using easypy as a job aggregator offers several benefits:

1. Centralized Job Execution: You can run multiple test scripts as part of a single job, making it easier to manage and execute tests across different scenarios or environments.
2. Consolidated Reporting: easypy aggregates the results from all test scripts and generates a consolidated test report, providing a comprehensive view of the job execution and its outcomes.
3. Parallel Execution: You can configure easypy to run multiple test scripts in parallel, leveraging the available resources and reducing the overall execution time.
4. Flexible Configuration: The job file allows you to define specific configurations or parameters for each test script, tailoring the execution to your needs.

By using easypy as a job aggregator, you can streamline the execution and reporting of multiple test scripts, making it easier to manage complex test suites and analyze the overall test results.

Network Journey
A journey towards packet life !!!

FIRST CREATE EASYPY JOB SCRIPT

EASYPY easypy <job_file.py>

cat 1easypy.py

```
# Import the Easypy module
from pyats.easypy import run

# The main function of the script
def main():
    # Run the job file
    run("1example.py")
```

refer next slide 1example.py , 2example.py, 3example.py

Network Journey
A journey towards packet life !!!

SCRIPT1

```
root@networkjourney-virtual-machine:/home/networkjourney/pyats-env/easypy# cat 1example.py
# Import the necessary modules
from pyats import aetest
import logging

class MyFirstTestcase(aetest.Testcase):
    """This is a very basic test case"""

    @aetest.setup
    def setup(self):
        """Test case setup"""
        logging.info("Setup Section")

    @aetest.test
    def test(self):
        """Our test section"""
        logging.info("Test Section")

    @aetest.cleanup
    def cleanup(self):
        """Clean up after the test"""
        logging.info("Cleanup Section")

if __name__ == '__main__':
    # We use aetest.main() to start the test
    aetest.main()
```

SCRIPT#2 cat 2example.py

```
# show_version_testcase.py

# import necessary modules
from pyats.aetest import Testcase, test
from genie.testbed import load

# define a testcase
class ShowVersionTestcase(Testcase):

    @test
    def fetch_show_version(self):
        # load the testbed file
        testbed = load('my_testbed.yaml')

        # select the device from the testbed
        device = testbed.devices['SW01']

        # connect to the device
        device.connect()

        # execute the 'show version' command and fetch the output
        output = device.execute('show ip interface brief')

        # print the output
        print(output)

        # disconnect from the device
        device.disconnect()
```

```
root@networkjourney-virtual-machine:/home/networkjourney/pyats-env/easypy# cat 3example.py
# show_version_testcase.py

# import necessary modules
from pyats.aetest import Testcase, test
from genie.testbed import load

# define the first testcase
class ShowVersionTestcase(Testcase):

    @test
    def fetch_show_version(self):
        # load the testbed file
        testbed = load('my_testbed.yaml')

        # select the device from the testbed
        device = testbed.devices['SW01']

        # connect to the device
        device.connect()

        # execute the 'show version' command and fetch the output
        output = device.execute('show version')

        # print the output
        print(output)

        # disconnect from the device
        device.disconnect()

# define the second testcase
class ShowInterfacesTestcase(Testcase):

    @test
    def fetch_show_interfaces(self):
        # load the testbed file
        testbed = load('my_testbed.yaml')

        # select the device from the testbed
        device = testbed.devices['SW01']

        # connect to the device
        device.connect()

        # execute the 'show interfaces' command and fetch the output
        output = device.execute('show interfaces')

        # print the output
        print(output)

        # disconnect from the device
        device.disconnect()
```

Network Journey
A journey towards packet life !!!

TEXTFSM

TextFSM is a Python module which implements a template-based state machine for parsing semi-formatted text. It was developed by Google and is often used in the networking community for parsing CLI output from network devices.

Documentation:
https://pyneng.readthedocs.io/en/latest/book/21_textfsm/README.html

Network Journey
A journey towards packet life !!!

how to write textfsm document:
https://pyneng.readthedocs.io/en/latest/book/21_textfsm/README.html

SCRIPT#1

```
from netmiko import ConnectHandler
import textfsm

# Define the device details
device = {
    'device_type': 'cisco_ios',
    'ip':   '192.168.61.144',
    'username': 'admin',
    'password': 'cisco',
}

# Connect to the device
connection = ConnectHandler(**device)

# Execute a command
output = connection.send_command('show ip interface brief')

# Load TextFSM template
template_file = 'cisco_ios_show_ip_interface_brief.textfsm'  # replace this with the path to your
template file
with open(template_file, 'r') as f:
    template = textfsm.TextFSM(f)

# Use TextFSM to parse the output
parsed_output = template.ParseText(output)

print(parsed_output)
print(parsed_output[0])
```

```
^^^^^^^^^^^
create in pycharm cisco_ios_show_ip_interface_brief.textfsm
^^^^^^^^^^^
Value List INTERFACE (\S+)
Value IP_ADDRESS (\S+)
Value STATUS (\S+)
Value PROTOCOL (\S+)

Start
 ^Interface\s+IP-Address\s+OK\?\s+Method\s+Status\s+Protocol -> IP_INT_BRIEF

IP_INT_BRIEF
 ^${INTERFACE}\s+${IP_ADDRESS}\s+\S+\s+\S+\s+${STATUS}\s+${PROTOCOL} -> Record
```

how to write textfsm document:
https://pyneng.readthedocs.io/en/latest/book/21_textfsm/README.html

SCRIPT#2

```
from netmiko import ConnectHandler
import textfsm

# Define the device details
device = {
    'device_type': 'cisco_ios',
    'ip':   '192.168.61.144',
    'username': 'admin',
    'password': 'cisco',
}

# Connect to the device
connection = ConnectHandler(**device)

# Execute a command
output = connection.send_command('show ip ospf interface')

# Load TextFSM template
template_file = 'cisco_ios_show_ip_interface_brief.textfsm'  # replace this with the path to your
template file
with open(template_file, 'r') as f:
    template = textfsm.TextFSM(f)

# Use TextFSM to parse the output
parsed_output = template.ParseText(output)

print(parsed_output)
```

^^^^^^^^^^^
create in pycharm cisco_ios_show_ip_interface_brief.textfsm
^^^^^^^^^^^

```
Value Required INTERFACE (\S+)
Value IP_ADDRESS (\S+)
Value AREA_ID (\S+)
Value PROCESS_ID (\d+)

Start
  ^${INTERFACE} is up, line protocol is up
  ^  Internet Address ${IP_ADDRESS}, Area ${AREA_ID}
  ^  Process ID ${PROCESS_ID}, -> Record
```

how to write textfsm document:
https://pyneng.readthedocs.io/en/latest/book/21_textfsm/README.html

SCRIPT#3

```
from netmiko import ConnectHandler
import textfsm

# Define the device details
device = {
    'device_type': 'cisco_ios',
    'ip':   '192.168.61.144',
    'username': 'admin',
    'password': 'cisco',
}

# Connect to the device
connection = ConnectHandler(**device)

# Execute a command
output = connection.send_command('show ip bgp summ')

# Load TextFSM template
template_file = 'cisco_ios_show_ip_interface_brief.textfsm'  # replace this with the path to your
template file
with open(template_file, 'r') as f:
    template = textfsm.TextFSM(f)

# Use TextFSM to parse the output
parsed_output = template.ParseText(output)

print(parsed_output)
```

```
^^^^^^^^^^^
create in pycharm cisco_ios_show_ip_interface_brief.textfsm
^^^^^^^^^^^
Value PEER (\S+)
Value VERSION (\d+)
Value ASN (\d+)
Value MSG_REC (\d+)
Value MSG_SENT (\d+)
Value STATE (\S+)

Start
  ^Neighbor\s+V\s+AS\s+MsgRcvd\s+MsgSent\s+TblVer\s+InQ\s+OutQ\s+Up/Down\s+State/PfxRcd\s*$$
  ^${PEER}\s+${VERSION}\s+${ASN}\s+${MSG_REC}\s+${MSG_SENT}\s+\S+\s+\S+\s+\S+\s+${STATE} -> Record
```

Handling Large Inventory using Multi-Threading with Asyncio, Threading

SCRIPT#1

```
import asyncio
import aiohttp

# Define your inventory as a list of URLs
inventory = ['http://example.com/item1', 'http://example.com/item2',
'http://example.com/item3']

async def get_item(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        tasks = []
        for url in inventory:
            tasks.append(get_item(session, url))
        await asyncio.gather(*tasks)

if __name__ == "__main__":
    asyncio.run(main())
```

Network Journey
A journey towards packet life !!!

KLEENEX \ PYATS CLEAN

Kleenex is a framework built on top of the Cisco pyATS/Genie automation framework. It provides a set of tools and features to facilitate network configuration and validation tasks. Here are some reasons why Kleenex is used:

1. Simplified Network Configuration: Kleenex simplifies the process of configuring network devices by providing a declarative and YAML-based syntax for defining network configurations. This allows for easier management of device configurations, reducing the need for manual command-line configurations.
2. Idempotent Configuration Management: Kleenex ensures that network configurations are idempotent, meaning that applying the same configuration multiple times has the same result as applying it once. This helps prevent configuration drift and ensures consistency across devices.
3. Configuration Validation: Kleenex enables the validation of network configurations against predefined rules and templates. It checks for configuration compliance, best practices, and potential issues, helping to ensure that configurations are correct and adhere to desired standards.
4. Pre- and Post-change Validation: Kleenex allows for pre- and post-change validations to verify the impact of configuration changes on the network. This helps in reducing the risk of misconfigurations and ensures that the network is operating as expected before and after changes are made.
5. Automated Network Testing: Kleenex provides the ability to automate network testing by defining test cases and scenarios. It supports the execution of various types of tests, such as performance testing, feature testing, and interoperability testing, to validate network behavior and performance.

Overall, Kleenex simplifies network configuration management, reduces manual efforts, ensures configuration consistency, and facilitates network validation and testing. It is designed to enhance network automation and operational efficiency.

Network Journey
A journey towards packet life !!!

Example to PYATS KLEENEX:
create factory_reset.yaml

```
devices:
  Device1:
    cleanup:
      - genie.libs.clean.stages.ios.factory_reset:
          arguments:
            reload_timeout: 500
```

```
pyats run genie cleanup --clean-file factory_reset.yaml --testbed-file testbed.yaml
```

Network Journey
A journey towards packet life !!!

Example to clear BGP configs:

```
#call library
from genie.conf import Genie
from genie.libs.sdk.apis.iosxe.bgp.remove import remove_all_bgp_config


# Initialize the testbed
testbed = Genie.init('testbed.yaml')

# Select a device
device = testbed.devices['SW01']

# Connect to the device
device.connect()

# Remove all BGP configurations
remove_all_bgp_config(device)

# Disconnect from the device
device.disconnect()
```

Network Journey
A journey towards packet life !!!

```
from pyats.topology import loader
# Load the testbed file
testbed = loader.load('testbed.yaml')
# Select a device
device = testbed.devices['SW01']
# Connect to the device
device.connect()
# Execute a configuration command to clear an interface
device.configure('default interface GigabitEthernet0/1')
# Or if you want to reset counters on the device
device.execute('clear counters')
# Disconnect from the device
device.disconnect()
```

Network Journey
A journey towards packet life !!!

Example using GENIE CLI method:

a. create clean-file ospf_clean.yaml

```
devices:
  Device1:
    cleanup:
      - genie.libs.clean.stages.iosxe.ospf:
          arguments:
            interfaces:
              - GigabitEthernet0/1
```

```
pyats run genie cleanup --clean-file ospf_clean.yaml --testbed-file testbed.yaml
```

Network Journey
A journey towards packet life !!!

TRAFFIC GENERATION SCENERIO:

Network Journey
A journey towards packet life !!!

```
from pyats import aetest
from pyats.topology.loader import load
from scapy.all import *

# Define your testbed
testbed = load('testbed.yml')

# Create a test section
class TrafficGeneratorTest(aetest.Testcase):

    # Define a test case to generate traffic
    @aetest.test
    def generate_traffic(self):
        # Access the device and interface to generate traffic
        device = self.parent.parameters['SW01']
        interface = self.parent.parameters['GigabitEthernet0/0']

        # Generate traffic using scapy
        packet = Ether(src='00:00:00:00:00:01', dst='00:00:00:00:00:02') / IP(src='172.16.24.21', dst='172.16.24.1') / ICMP()

        # Send the packet on the specified interface
        sendp(packet, iface=device.interfaces[interface].name)

        self.passed('Traffic generated successfully')

# Create an instance of the test section
test = aetest.Testbed(aetest.GenericTestbed)
test.add_device(testbed.devices['SW01'])

# Add additional parameters if needed, such as the interface to generate traffic on
test.parameters['interface'] = 'GigabitEthernet1/0/1'

# Run the test
aetest.main(testbed=testbed)
```

Network Journey
A journey towards packet life !!!

PORT VALIDATION SCENERIO:

Network Journey
A journey towards packet life !!!

```
from pyats import aetest
from pyats.topology.loader import load
from pyats.topology import interfaces

# Define your testbed
testbed = load('testbed.yaml')
multiple = ["SW01","SW02"]
for singledevice in multiple:
 # Create a test section
 class PortValidationTest(aetest.Testcase):
     @aetest.test
     def validate_port(self):
         # Access the device and port to validate
         device = self.parent.parameters[singledevice]
         port = self.parent.parameters['25']

         # Validate the port
         is_valid = interfaces.validate_interface_name(port)

         # Check the validation result
         if is_valid:
             self.passed(f"The port {port} is valid.")
         else:
             self.failed(f"The port {port} is invalid.")



 # Create an instance of the test section
 test = aetest.Testbed(aetest.GenericTestbed)
 test.add_device(testbed.devices[singledevice])

 # Add additional parameters if needed, such as the port to validate
 test.parameters['port'] = 'GigabitEthernet0/1'


 # Run the test
 aetest.main(testbed=test)
```

Network Journey
A journey towards packet life !!!

TEACAT BUGS AUTOMATION:

Network Journey
A journey towards packet life !!!

```
from pyats import aetest
from pyats.topology.loader import load
import datetime

# Define your testbed
testbed = load('testbed.yml')

# Create a test section for bug automation
class BugAutomationTest111111111111111(aetest.Testcase):

    @aetest.test
    def report_bug(self):
        # Access the device and relevant information for the bug report
        device = self.parent.parameters['SW01']
        bug_title = self.parent.parameters['bug_title']
        bug_description = self.parent.parameters['bug_description']

        # Generate a timestamp for the bug report
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        # Construct the bug report
        bug_report = f"Bug Title: {bug_title}\n"
        bug_report += f"Device: {device}\n"
        bug_report += f"Timestamp: {timestamp}\n"
        bug_report += f"Description: {bug_description}\n"

        # Submit the bug report or save it to a file
        # Example: Write the bug report to a text file
        with open('bug_report.txt', 'w') as file:
            file.write(bug_report)

        self.passed('Bug report generated successfully')

# Create an instance of the test section
test = aetest.Testbed(aetest.GenericTestbed)
test.add_device(testbed.devices['SW01'])

# Add additional parameters for the bug report
test.parameters['bug_title'] = 'Bug in Network Configuration'
test.parameters['bug_description'] = 'The network configuration is not applied correctly.'
test.parameters['timestamp'] = datetime.datetime.now().strftime("%Y-%d %H:%M:%S")

# Run the test
aetest.main(testbed=test)
```

Network Journey
A journey towards packet life !!!

THE END

Thank you being good students throughout the sessions for last 20 days

All Best for Your Future Career