Introduction

Learn network programmability with Python, GNS3, and Cisco devices.

What you'll learn

1. Python fundamentals
2. Network automation with Python

About the author

I am an Associate Engineer (DAE in Electronics) exploring network automation as a hobby after working in computer networks for 25 years. My skills as an Associate Engineer include:

- Routing and Switching
- OFC/LAN Networking
- IP Addressing and Sub-netting
- Computer Basics - Windows 7/10
- Linux and Ubuntu Desktop/Server
- VMware/KVM/VirtualBox
- Docker/Vagrant - Hands On
- Ansible for Network Automation

Computer Programming - An Introduction

Inside every computer, there's a special set of instructions that makes a computer to work, is called a computer program. It's the essential life force that transforms computer hardware into a functional device. To help you understand this concept, think of a computer as a piano, an instrument that remains silent without a skilled musician.

However, computers, by their nature, are exceptionally proficient at executing some basic operations, such as addition and division, etc. A computer is performing these operations at lightning speed and with flawless accuracy.

Now, taking into a practical scenario. Imagine you're on a long road trip, and you want to calculate your average speed. The distance you've traveled and the time it took to reach your destination, but here's the problem, that computers don't intuitively understand these concepts as a human you do. Therefore, you need to provide precise instructions to the computer, as instructing it to:

Python for Network Engineers

Python, a versatile and powerful programming language, has become a must-have tool for network engineers. Whether you're just starting in networking or an experienced pro, Python offers a wealth of benefits for network-related tasks. Let's explore these key aspects and guide you through the basics of Python for network engineering.

Network Automation: It helps network engineers automate repetitive tasks such as making configuration changes, creating backups, and monitoring networks.

Configuration Management: Python-based tools, including Ansible, NAPALM, and Netmiko, are widely used for configuration management.

Monitoring and Troubleshooting: Python allows network engineers to create custom monitoring and troubleshooting tools that continuously check the health of a network.

Network Security: Python plays a vital role in implementing

Understanding Python String

Python is a versatile programming language, offers a wide range of data types, including strings, numbers, lists, dictionaries, and more. In this blog, we'll focus on the fundamental data type known as a *string*.

A string in Python is a sequence of characters, such as letters, numbers, and symbols, enclosed within single or double quotation marks. It allows you to work with textual data, making it an essential component for tasks like text processing, data manipulation, and user interactions. Strings can be combined, sliced, modified, and processed in numerous ways, making them a crucial element in any Python program.

Creating String

In Python, strings are a fundamental data type used to work with textual information. You can create strings using both single and double quotes. For example, you can define a string like this:

```
Python 3.10.7 ..............  [MSC v.1933 64 bit (AMD64)] on
```

Understanding Python Numbers

Numbers in Python are a fundamental data type used for various mathematical operations and calculations. There are primarily two numeric data types in Python: integers (int) and floating-point numbers (float).

1. *Integers Numbers (int):* Integers are typically used when working with discrete values or countable objects. They can be positive, negative, or zero. For example, 5, -10, and 0 are all integers in Python. Integers are typically used when you need to work with discrete values or countable objects.

2. *Floating-Point Numbers (float):* Floating-point numbers, or floats, are numbers that include a decimal point or use scientific notation, such as 3.14 or 2.5e-3. Floats are used when you need to work with real numbers, including fractional values and approximate calculations.

These two numeric data types are essential for handling a wide range of mathematical and numerical operations in Python, making it a versatile language for tasks involving arithmetic and

Understanding Lists in Python

A list in Python is a fundamental data structure that allows you to store a collection of items. Lists are versatile and can hold various types of data, including strings, integers, booleans, other lists, and more. In this blog, we'll dive into the world of Python lists and cover everything you need to know.

List Basics

A list in Python is a collection of items. Here are some key characteristics of lists:

- Lists maintain the order in which elements are added. This means you can access elements by their position within the list.
- Lists can hold a mix of different data types. For example, you can have a list that contains strings, integers, booleans, and even other lists.
- Lists are mutable, which means you can change their elements, size, and structure during program execution.

These characteristics makes lists a fundamental and dynamic data structure for various tasks. In some other programming

Understanding Mutable and Immutable Objects in Python

Python, is a popular and versatile programming language, the distinction between mutable and immutable objects is key concept to comprehend for effective Python programming. This concept significantly impacts how Python behaves in various situations. In this blog, we'll explore these concepts step by step, using code examples to make it clear.

What Are Mutable and Immutable Objects?

In Python, we classify objects into two categories: mutable and immutable. Mutable objects can change their values after creation, while immutable objects stay the same once created.

How Assignments Work

Let's start by understanding how assignments function in Python. When you assign a value to a variable, like `rtr_addr = "10.1.1.1"`, Python allocates memory to store that value, and the variable `rtr_addr` becomes a reference to that memory location.

Understanding Tuples in Python

In Python, a tuple is a data structure that is used to store an ordered collection of items. Tuples are similar to lists, but they have a few key differences. This blog post will provide a comprehensive explanation of tuples in Python, covering their definition, characteristics, and common use cases.

What is a Tuple?

A tuple is an ordered collection of items that can contain elements of different data types. Tuples are defined using parentheses `()`, and the elements inside a tuple are separated by commas. Here's an example of creating a tuple:

```
my_tuple = (1, "hello", 22, None, 2.7)
```

Storing Different Data Types in a Tuple

Tuples can store elements of different data types. In the example above, `my_tuple` contains integers, a string, `None`, and a floating-point number. This flexibility makes tuples versatile for various use cases.

Using Parentheses

Conditional Statements in Python

In Python, conditional statements play a pivotal role in controlling the flow of your program. They allow you to execute specific blocks of code based on the truth value (True or False) of certain conditions.

An **expression** is a fundamental concept in Python, representing a piece of code that can be evaluated to produce a value. Expressions typically involve variables, constants, and operators. Conditions, which determine the execution of code blocks, are constructed using these expressions.

Let's delve into the key conditional statements in Python:

if statement: This fundamental construct enables you to execute a block of code when a given condition evaluates to True. It's a core element of Python's conditional logic.

elif statement: When you need to evaluate multiple conditions one after another, the `elif` statement comes into play. It allows you to check a series of conditions, and when one of them proves

Understanding Booleans

Booleans in Python are a fundamental data type that represents two values: `True` and `False`. Booleans are case-sensitive in Python, so `True` and `False` must be written with an uppercase initial letter. Using lowercase such as `true`, or `false`, will result in a NameError.

You can use the `type()` function to check the data type of a variable, including Boolean variables. For example, if you want to check if a variable is a Boolean, you can do the following:

```
Python 3.10.7 ..... [MSC v.1933 64 bit (AMD64)] on
Type "help", "copyright", "credits" or "license" for more
>>> my_variable = True
>>> type(my_variable)
<class 'bool'>
```

This code will correctly identify `my_variable` as a boolean and print `<class 'bool'>`.

Boolean Logic in Python

Working with Files in Python

Working with files in Python, is a common task, and it's essential to understand how to read and manipulate the contents of files. In this guide, we'll delve into file handling, beginning with the fundamental method, and then we'll explore alternative approaches for file reading and writing.

File Reading in Python

File reading in Python allows you to access the contents of a file. The basic method for reading a file is as follows:

```
f = open('show_version.txt')
data = f.read()
f.close()
```

Let's break down the code step by step:

Opening the File: The `open()` function is used to open a file. In this example, 'show_version.txt' represents the file you intend to read. If the file is located in the same directory as your Python script, you can simply provide the filename. The `open` function returns a file object, which we assign to the variable `f`, commonly referred to as a file handler.

Understating For Loop in Python

In Python, there are different ways to iterate over collections or sequences, and one of the most common methods is the `for` loop. The `for` loop allows you to execute a block of code for each item in an iterable, such as a list, string, tuple, set, or dictionary. Let's break down how a `for` loop works using the example:

```
ip_list = ["10.88.17.1", "10.88.17.2", "10.88.17.20", "10.8
for ip in ip_list:
    print(ip)
```

Here's a breakdown of the key components and how they work:

1. **`for` keyword**: The `for` keyword initiates the loop. It tells Python that you want to start a loop.

2. **Loop variable (`ip`)**: In the line `for ip in ip_list:`, `ip` is the loop variable. It's a temporary variable that represents each item in the iterable during each iteration of the loop.

3. **Looping object (`ip_list`)**: The `ip_list` is the list that you want to loop over. In each iteration of the loop, the loop

While Loop in Python

A while loop is another powerful construct in Python used to repeatedly execute a block of code as long as a certain condition remains true. It's a valuable tool when you want to execute code based on a specific condition, which might not be known beforehand.

A while loop consists of an expression followed by a colon and an indented block of code. The loop continues executing as long as the expression evaluates to `True`.

```
while expression:
    print("message")
```

This code will repeatedly print "message" as long as the `expression` remains true.

A common pitfall with while loops is creating an infinite loop, where the loop never exits. To avoid this, always ensure that the expression within the while loop will eventually become false, or use the `break` keyword to exit the loop.

List Comprehensions: Simplifying Data Manipulation

List comprehensions are a powerful tool that makes working with lists in Python more efficient and concise. They are especially valuable for network engineers and Python enthusiasts. In this article, we'll dive into list comprehensions, covering the basics and sharing tips and best practices to help you unlock their potential.

Understanding List Comprehensions

List comprehensions provide a straightforward way to create lists in Python. They follow a specific structure:

- **Syntax**: `[expression for item in iterable]`
- The square brackets signify that we're creating a list.
- `expression` is the value to include in the list for each `item` in the `iterable`.
- `item` represents the current element in the `iterable`.

To illustrate, here are some basic examples:

Example: Creating a List of Squares

```
squares = [x**2 for x in range(1, 6)]
```

Understanding Sets in Python

Sets are a powerful and versatile data structure in Python that can handle collections of distinct elements. Unlike other data structures, such as lists or tuples, sets are unordered and mutable, meaning that you can modify their content after creation. Sets also support various operations, such as union, difference, and intersection, that mimic mathematical set operations. In this tutorial, you will learn how to create and manipulate sets in Python using different methods and examples. You will also discover how sets can be useful for network engineering tasks, such as managing unique IP addresses, VLAN IDs, or network devices.

How to Use Sets in Python

Sets are a data type in Python that store collections of unique elements. They are useful for working with distinct items, such as IP addresses or network devices.

Creating Sets

To create a set, we use curly braces `{}` and commas to separate

Set Comprehensions in Python

Python, renowned for its versatile Python programming and diverse data structures, hosts a significant asset: Set Comprehension. This capability enables dynamic set creation from iterable objects, particularly valuable in network automation scenarios.

Understanding Set Comprehensions

Set Comprehensions, similar to List Comprehension, stand out for their syntax encapsulated within curly braces. They enable the creation of sets from various iterables, pivotal in managing network data.

For instance, with a set of device identifiers in a network, leveraging Set Comprehension simplifies creating a unique set of devices:

```
network_devices = {"router1", "switch1", "router2", "firewa
unique_devices = {device for device in network_devices}
print(unique_devices)
# {"router1", "switch1", "router2", "firewall1", "switch2"
```

This example emphasizes the efficiency of sets in managing

Mastering Python Dictionaries: A Network Engineer's Guide

Dictionaries are versatile data structures that store information as key-value pairs. They maintain the order of items and are useful for various programming tasks. You can create dictionaries using curly braces {} and access values by keys. Python 3.7 onwards, dictionaries are ordered by default, making them even more powerful for efficient and organized code.

Dictionaries are like magical containers that hold **key-value pairs**. Each key corresponds to a specific value, allowing you to organize and retrieve information efficiently. Here's how you create one:

```
my_dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'val
# Output: {'key1': 'value1', 'key2': 'value2', 'key3': 'va
```

In this example, `'key1'` maps to `'value1'`, `'key2'` to `'value2'`, and so on. These curly braces `{}` enclose the dictionary, making it a powerful tool for Python programmers.

Functions and Classes

In Python, the concept of functions plays a crucial role, allowing developers to encapsulate and reuse code efficiently. Functions provide a way to create modular and maintainable code, as specific tasks can be performed by calling a function multiple times, eliminating the need for redundant code.

Python's strength in object-oriented programming (OOP) further enhances its capabilities. In Python, almost everything is treated as an object, each associated with functions (methods), properties (attributes), and variables. A class serves as a blueprint in OOP, defining the structure and behavior of objects. Instances of a class can be created, essentially representing concrete examples based on the class's specifications.

When naming functions, objects, and modules in Python, adhere to the convention of using lowercase letters separated by underscores. For instance:

```
def my_router():
    pass
```

Setting Up a Network Lab with GNS3

Introduction

Networking labs, providing a sandbox for testing, learning, and refining skills. In this blog, we'll delve into the process of setting up a virtual network lab using GNS3 on a Windows PC. This comprehensive guide will walk you through the installation of GNS3 and its virtual machine (GNS3 VM) within VMware. Additionally, we'll explore the integration of an Ubuntu 22 Server, establishing a robust foundation for hands-on networking experimentation.

The key components of this setup include GNS3, a powerful network simulation tool, VMware for virtualization, and an Ubuntu Server for practical application testing. By following the outlined steps, you'll create a cohesive environment that bridges your Windows host, GNS3, and Ubuntu Server, fostering seamless interaction.

This introductory section sets the stage for an insightful journey into the intricacies of building a dynamic virtual network lab. As we progress through the subsequent sections, we'll cover each

Telnet Programming in Python: Streamlining Network Operations

Telnet, an abbreviation for Telecommunication Network, stands out as a protocol that facilitates text-based communication sessions with remote devices over a network. Operating at the application layer of the OSI model, Telnet proves invaluable for accessing and managing network devices. In contrast to graphical interfaces, Telnet relies on plain text communication, offering a lightweight and versatile option for various networking scenarios.

Understanding Telnet's Operation

Telnet operates on a client-server model where the client initiates a connection to the server. Once connected, the client can send commands, and the server responds with the output, typically in ASCII format, making it human-readable.

Importance in Network Programming

Empowering Network Automation

Telnet plays a pivotal role in network automation by enabling

Streamlining Network Operations with Python and Paramiko

When it comes to network automation, Python stands out as a versatile language, and Paramiko, a Python library, takes the center stage in simplifying secure communication over SSH. In this blog post, we'll explore the capabilities of Paramiko through practical examples that showcase its effectiveness in automating network configurations.

Getting Started with Paramiko

Paramiko serves as a robust implementation of the SSHv2 protocol, offering both client and server functionality. It plays a crucial role in the toolkit of network engineers, enabling seamless automation of tasks on networking devices. Let's delve into the core features and a simple example of its application.

Installation

Before diving into the examples, ensure that you have Paramiko installed. You can do this effortlessly using the following pip command:

Appendix A: Additional Resources

This section contains additional resources related to the content of the book.

Pandoc Command

Use this command to create a pdf book.

```
pandoc chapters/*.md -o output.pdf \
--metadata-file=metadata.yml \
--resource-path=. --toc --number-sections \
--include-in-header main.tex \
--highlight-style pygments.theme \
-V geometry:a5paper \
-V geometry:margin=2cm \
-V fontsize=12pt
```

Pypandoc

Pypandoc provides a thin wrapper for pandoc, a universal document converter.

```
import os
import pypandoc
```