

## Python for IT Automation

### 2.1 Intro to Python

- High level, general purpose programming language
- Go-to for automating networking and cloud configurations
- Simple syntax makes it easy to learn
- Scripting
  - Automates repetitive tasks
    - Configuring multiple routers simultaneously
    - Assigning permissions to a large group of users
    - Creating ad hoc virtual networks
    - Handling security events and notices

### 2.2 What is Python?

- Programming Paradigm
  - A fundamental style or approach to programming that dictates how tasks should be structures and executed in each programming language
  - Types of Programming
    - Imperative/Procedural
      - Describes a sequence of steps to perform a task
    - OOP
      - Organizes code around objects, which encapsulate data and behavior
    - Functional
      - Emphasizes the use of functions and immutable data for computation
    - Event-driven
      - Reacts to events and user actions, triggering corresponding functions
    - Logic
      - Defines a set of logical conditions and lets the system deduce solutions

### 2.3 Python Syntax

- Indentation
  - One indent is 4 spaces
  - Used to determine the structure and nesting of code blocks
- Comments
  - # is used to write one lined comments
- Variables
  - `x = 10`
  - Types

- No need for declaration
- Assign a value to a name and python figures out the type
- Example
  - number = 42
  - text = "router configuration"
  - ipAddress = "10.200.30.254"
  - my\_list = [router, switch, firewall]
- Conditional Statements
  - if, elif, else
    - Used with a ":" at the end
    - The next indentation's code block is part of the if, elif, or else
- Loops
  - Used for iterative execution of code
  - for i in range(5):
    - print i
  - while x > 0:
    - print(x)
    - x -= 1
- Classes
  - class CiscoRouter:
    - def \_\_init\_\_(self, name):
      - self.name = name
    - def config(self)
      - print("Here is the config for this router")
- print() Statement
  - print("This is the router configuration")
  - automatically adds a new line for each print statement
- Functions
  - def greet(name):
    - print("Hello, " + name + "!!")
- Importing modules
  - import os
  - import paramiko

## 2.4 Executing Python Code

- Command Prompt
  - Navigate to where your file is located and enter:
    - python filename.py
    - python3 filename.py
- IDE
  - Visual Studio Code from Microsoft, PyCharm, etc.

- Interactive Mode
  - typing in “python” into the terminal will allow for python to be written and run interactively line by line

## 2.5 Python Comments

- Use for collaboration, teaching, and troubleshooting

## 2.6 Whitespace

- Understand indentation and previous covered
- Python solely relies on indentation for defining code blocks
  - No brackets, braces, or parenthesis needed
  - Makes code much more clean
  - Makes Python “indentation sensitive”

## 2.7 Blocks and Scope

- Python uses indentation for block delimitation
  - Block group code associated with control statements like if
  - Indentation levels define block hierarchy
  - Changes in indentation determine variable scope; variables defined within maintain scope outside
  - Variable scope is influenced by functions, objects, and modules, not blocks

## 2.8 Python Input Output

- Input
  - Captured using the input() function, storing it as string
- Output
  - Displayed with the print() function
- Format Method
  - format() method enhances output formatting by embedding variables in strings
- File I/O
  - open() function

## 2.9 Built-In Functions

- print()
  - prints the following text onto the string
  - use (f“{var}”) to add variables into the print function
- input()
  - Stops the code and asks for input, with the text in the parentheses being the prompt
- len()
  - determines the length of a sequence (e.g., string, list, tuple)
- type()
- int(), float(), str()
  - use to convert values into ints, floats, or strings
- max(), min()

- returns the max or min from a sequence
- `sum()`
  - sums all elements in a sequence
- `abs()`
  - returns the absolute value
- `range()`
  - generates a sequence of numbers
- `sorted()`
  - returns a sorted list from an iterable
- `any()`, `all()`
  - checks if any or all elements in an iterable are true
- `map()`, `filter()`
  - applies a function to elements or filters elements based on a function
- `open()`, `read()`, `write()`
  - handles file I/O operations
- `dir()`
  - lists the names in the current scope or attributes of an object
- `help()`
  - provides help information about an object or Python

## 2.10 Python IDE

- code editor
  - A text editor designed for Python, offering features like syntax highlighting, code completion, and indentation
- debugger
  - Allows developers to identify and fix errors in their code by providing tools for step-by-step execution and variable inspection
- interactive shell
  - Enables the execution of Python code interactively, providing a command-line interface for testing snippets of code
- project management
  - Helps organize and manage Python projects by providing tools for creating, opening, and saving projects
- version control integration
  - Supports integration with version control systems like Git for efficient collaboration and code versioning
- code navigation
  - Facilitates easy navigation through large codebases by offering features such as code folding, code outlining, and quick navigation to definitions
- integrated documentation

- Provides access to documentation for Python libraries and modules directly within the IDE
- code profiling
  - Allows developers to analyze code performance and identify areas for optimization

## 2.11 Problem Solving

- 4 Step Problem Solving Process
  - Analyze
  - Develop Algorithm or Pseudocode
  - Writing Code
  - Testing and Debugging

## 3.2 Variables

- Are NOT declared
- A variable is created as soon as you assign a value to the variable
  - `x = 1.1234 #float`
- Can be changed after being created
  - `x = 42 #integer`
  - `x = "hello" #now a string`
- Case Sensitive
  - `X = "hello" #is not the same as`
  - `x = "hello"`
- Variable Rules
  - Can only contain letters, numbers, and underscores
  - Can only start with a letter or underscore
  - Cannot start with a number
  - Cannot contain special characters
  - Examples of valid names
    - `switchname_e`
    - `cisco2093`
    - `y`
    - `y2`
  - Examples of NON-valid names
    - `2variab`
    - `$hi`
    - `.hello`

## 3.3 Multiple Variables

- Example of single line declaration
  - `x, y, z = "router", "switch", "firewall"`
  - # the number variables on the left must match the right
- Example of single line declaration to the same value

- `x = y = z = "hello"`
- Unpacking Collections
  - Example 1
    - `ip_addresses = ["172.20.200.10", "172.20.200.11", "172.20.200.12"]`
    - `ip1, ip2, ip3 = ip_addresses`
  - Example 2
    - `device_information = ["laptop1", "10.0.0.1", "255.255.255.0"]`
    - `name, ip_address, subnet_mask = device_information`

### 3.4 Output Variables

- Example of combining using commas and concatenation
  - `ip1 = "10.100.200.30"`
  - `ip2 = "10.100.200.40"`
  - `ip3 = "10.100.200.50"`
  - `print(ip1, ip2, ip3)`
  - `print(ip1 + ip2 + ip3)`
  - 
  - `10.100.200.30 10.100.200.40 10.100.200.50`
  - `10.100.200.3010.100.200.4010.100.200.50`
- Adding number variables in python using +
  - `x = 15`
  - `y = 9`
  - `print(x + y)`
- Printing a number and a string in python
  - Must use commas, + will not work
  - Example
    - `x = 15`
    - `y = "server"`
    - `print(x, y)`

### 3.5 Variable Scope

- Inside a Function
  - Local Scope
    - Can only be accessed where it was called
- Outside a Function
  - Global Scope
    - Can be accessed from any part of the program, even functions
- Find the difference
  - Example 1
    - `ip1 = "10.2.60.5"`
    - `def myFunction():`
      - `print("The workstation IP address is " + ip1)`

- myFunction()
- Example 2
  - ip1 = "192.168.1.1"
  - def myFunction():
    - ip1 = "172.17.1.1"
    - print("This IP address is: " + ip1)
  - myFunction()
- 
- Global Keyword
  - You can use global before a variable in a function to make it global scope
  - Example
    - def makingGlobalVar():
      - global globalVar
      - globalVar = "10.10.10.129"
  - To change the value of a global variable inside of a function permanently, you need to declare the variable as global within the function

#### 4.1 Python Data Types

- Numeric Types
  - int and float
- Text Types
  - string
    - can use "" and ''
- Boolean Types
  - bool
    - true or false
- Sequence
  - list
    - ordered, mutable sequence
    - [1, 2, 3]
  - tuple
    - ordered, immutable sequence
    - (1, 2, 3)
  - range
    - represents a range of values
    - range(3)
- Set Types
  - set
    - unordered, mutable collection of unique elements
    - my\_set = {1, 2, 3}
  - frozenset

- unchangeable set
- Mapping Type
  - dict
    - dictionary, an unordered collection of key-value pairs
    - `my_dict = {'key': 'value', 'name': 'John'} # dict`
- None Type
  - none
    - represents the absence of a value or a null value
    - `my_variable = None`

#### 4.3 Number Data Types

- int
  - whole numbers without decimal points, that can be either positive or negative
  - 5, -10, 1000, 0
- floating-point numbers (float)
  - numbers with decimal points or in scientific notation
  - 3.14159, -0.065, 2.01e3
  - Allows representation of a broader range of values than integers
- int and float math
  - answer will convert to a float number
  - `int(numberhere)` will always round down
- round
  - `round(number)` will round normally
  - `round(number, num_of_decimals)` will round to that many decimal places
- abs
  - will get the absolute value of a int or float with no conversion happening
- dividing
  - / will result in a float if needed
- Example
  - get the average of the following 3 numbers rounded to the 2nd decimal
  - a, b, c = 5, 6, 9
  - `total = a + b + c`
  - `rounded_average = round((total / 3), 2)`

#### 4.4 Strings

- str
- uses single or double quotes
  - `greeting = 'hello'`
  - `goodbye = "bye bye"`
- create multiline strings using `''' text '''` (3 singles) or `""" text """` (3 doubles)
  - `serverIPAddresses = """Server IP Addresses are:`
  - 172.18.100.10,



- 172.18.100.11,
- 172.18.100.12,
- 172.18.100.13"""
- print(serverIPAddresses)
- String Operations
  - +
    - combine two strings
  - \*
    - repeating a string multiple times
  - []
    - accessing individual characters by position
    - 0 is the first position because strings are arrays in Python
- String Looping
  - for x in "string looping":
    - print x
- Formatted Strings
  - known as f-strings
  - message = f"Hello, my name is {name} and I am {age} years old."
    - with the variable inside of the brackets being converted to their value
- Escape Characters
  - \n
    - new line
      - multiline\_str = "This is a\nmultiline\nstring."
  - \t
    - insert tab
      - tabbed\_str = "This is a\ttabbed\tstring."
- String Methods
  - length = len(language) # Length of the string
  - uppercase\_str = language.upper() # Convert to uppercase
  - lowercase\_str = language.lower() # Convert to lowercase
  - new\_str = language.replace('P', 'J') # Replace characters
    - replace P with J
  - word\_list = language.split('t') # Split into a list
    - split at every instance of t
- Checking Strings
  - use "in" to check if a word is in a string
    - strText = "The Python language can be used in network automation tasks"
    - print("automation" in strText)
  - use "not in" to check if a word is not in a string
    - strText = "The Python language can be used in network automation tasks"

- `print("error" not in strText)`

#### 4.5 String Slicing

- Follows this format
  - `string[start:stop:step]`
    - start: The index from which the slicing begins (inclusive)
    - stop: The index at which the slicing ends (exclusive)
    - step (optional): The step or stride between characters
  - Defaults:
    - If start is not provided, it defaults to the beginning of the string.
    - If stop is not provided, it defaults to the end of the string.
    - If step is not provided, it defaults to 1.
  - Examples
    - `text = "Hello, Python!"`
    - # Extracting a substring from index 7 to the end
    - `substring = text[7:]`
    - `print(substring)` # Output: "Python!"
  
    - # Extracting the first 5 characters
    - `first_five = text[:5]`
    - `print(first_five)` # Output: "Hello"
  
    - # Extracting characters with a step of 2
    - `every_second = text[::2]`
    - `print(every_second)` # Output: "Hlo yhn"
  
    - # Reversing a string **VERY IMPORTANT**
    - `reversed_str = text[::-1]`
    - `print(reversed_str)` # Output: "!nohtyP ,olleH"

#### 4.6 Modify Strings

- String Length
  - use `len(string_name)` to find the length of a string
- Converting Characters
  - Uppercase
    - `string_name.upper()`
  - Lowercase
    - `string_name.lower()`
- Title
  - The `title()` method converts the first character of each word to uppercase.
    - `text = "hello python programming"`
    - `title_text = text.title()`

- `print(title_text) # Output: "Hello Python Programming"`
- Capitalize
  - The `capitalize()` method capitalizes the first character of a string.
  - `text = "hello, python!"`
  - `capitalized_text = text.capitalize()`
  - `print(capitalized_text) # Output: "Hello, python!"`
- Counting Substrings
  - The `count()` method counts the occurrences of a substring in the given string.
  - `text = "python is powerful, python is easy"`
  - `count_python = text.count("python")`
  - `print(count_python) # Output: 2`
- Replacing Substrings
  - The `replace()` method replaces occurrences of a specified substring with another string.
  - `text = "Python is easy and Python is fun"`
  - `new_text = text.replace("Python", "Java")`
  - `print(new_text)`
  - `# Output: "Java is easy and Java is fun"`
- Removing Whitespace
  - The `strip()`: method removes leading and trailing whitespaces, the `lstrip()` method removes leading whitespaces, and the `rstrip()`: method removes trailing whitespaces.
  - `text = " Python is easy "`
  - `stripped_text = text.strip()`
  - `print(stripped_text) # Output: "Python is easy"`
- Example Problem
  - Use built-in methods for string manipulation to print out the user message with the first four characters uppercase and the rest of the message lowercase.
  - 
  - `message = input("Enter a phrase: ")`
  - 
  - `first4 = message[:4] #grabs first 4 letters`
  - `first4_upper = first4.upper()`
  - `rest_of_message_lower = (message[4:]).lower()`
  - `message_first4_upper = first4_upper + rest_of_message_lower`
  - 
  - `print(message_first4_upper)`
  - `# Expected output for "Modern Networking"`
  - `# MODERN networking`

#### 4.7 Concatenate Strings

- Using the + Operator:
  - first\_name = "John"
  - last\_name = "Doe"
  - full\_name = first\_name + " " + last\_name
  - print(full\_name) # Output: "John Doe"
- Using the += operator
  - greeting = "Hello, "
  - greeting += "Python!"
  - print(greeting) # Output: "Hello, Python!"
- Joining Strings using join() method
  - The join() method is used to concatenate strings from an iterable (e.g., a list).
    - words = ["Python", "is", "awesome"]
    - sentence = " ".join(words)
    - print(sentence) # Output: "Python is awesome"
- Formatted String (f-string)
  - f-strings provide a concise way to embed expressions in strings.
    - item = "book"
    - quantity = 3
    - order\_summary = f"You ordered {quantity} {item}s."
    - print(order\_summary)
    - # Output: "You ordered 3 books."

#### 4.8 Formatting Strings

- Basic String Formatting (Positional Arguments)
  - The arguments are passed in order as variables
    - name = "John"
    - age = 25
    - # Using format() method with placeholders
    - message = "My name is {} and I am {} years old.".format(name, age)
    - print(message)
    - # Output: "My name is John and I am 25 years old."
- Index Based Formatting
  - The arguments coincide with the index shown within the brackets
    - product = "Laptop"
    - price = 1200.50
    - # Positional arguments in format()
    - invoice = "Product: {0}, Price: \${1}".format(product, price)
    - print(invoice)
    - # Output: "Product: Laptop, Price: \$1200.5"
- Name Based Formatting
  - You might as well enter in the values

- # Named arguments in format()
  - details = "Name: {name}, Age: {age}".format(name="Alice", age=30)
  - print(details)
  - # Output: "Name: Alice, Age: 30"
- Number Formatting
  - Controls how numbers are displayed
    - # Number formatting
    - value = 1234567.89
    - formatted\_value = "Formatted Value: {:.2f}".format(value)
    - print(formatted\_value)
    - # Output: "Formatted Value: 1,234,567.89"
  - Example with PI to 2 decimals
    - # Number formatting
    - value = 3.14159
    - formatted\_value = "Formatted Value: {:.3f}".format(value)
    - print(formatted\_value)
    - # Output: "Formatted Value: 3.142"
- Padding and Alignment
  - Padding:
    - Padding refers to adding characters to a string to reach a certain length. This is often used for output formatting and alignment.
    - There are three common types of padding:
      - Left Padding: Characters are added to the start of the string.
      - Right Padding: Characters are added to the end of the string.
      - Center Padding: Characters are added to both ends of the string until it reaches a specified length.
  - Alignment:
    - Alignment refers to the way text is positioned within a given space. In Python, you can align strings using the format() method with alignment operators:
    - 3 Types
      - < : Forces the field to be left-aligned within the width.
      - > : Forces the field to be right-aligned within the width.
      - ^ : Forces the field to be centered within the width.
  - Example
    - # Padding and alignment
    - text = "Python"
    - formatted\_text = "{:^10}".format(text)
    - print(formatted\_text)
    - # Output: " Python "

- Using Multiple Times
  - Example
    - # Reusing values
    - data = "{0} is a {1}. {0} is also {2}.".format("Python", "programming language", "fun")
    - print(data)
    - # Output: "Python is a programming language. Python is also fun."
- Python Booleans
  - True and False are the two Boolean literals in Python.
    - Example:
      - is\_python\_fun = True
      - is\_java\_fun = False
  - Logical Operators
    - Boolean values are often used with logical operators (and, or, not) for making logical decisions.
      - x = True
      - y = False
      - result\_and = x and y # False
      - result\_or = x or y # True
      - result\_not = not x # False
  - Comparison Operators
    - Comparison operators (==, !=, <, >, <=, >=) return Boolean values.
      - age = 25
      - is\_adult = age >= 18 # True
      - is\_teenager = age < 18 # False
  - Boolean Conversion
    - Boolean values can be explicitly converted from other types using the bool() function
      - number = 0
      - bool\_from\_number = bool(number) # False
    - In Python, values like non-zero numbers and non-empty strings are considered truthy, while zero, None, and empty strings are considered falsy

## Python for IT Automation

### 5.2 Python Operators

- In Python, operators are symbols that perform operations on variables and values.
  - Arithmetic operators (+, -, \*, /) handle numeric calculations.
    - +, -, \*, /, % (floor division), \*\* (exponentiation).
  - Comparison operators (==, !=, <, >, <=, >=) compare values, yielding Boolean results.

- `==` (equal), `!=` (not equal), `<`, `>`, `<=`, `>=`.
- Logical operators (and, or, not) perform logical operations on Boolean values.
  - and, or, not.
- Assignment operators (`=`, `+=`, `-=`, `*=`, `/=`) assign values to variables.
  - `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `//=`, `**=`.
- Membership operators (in, not in) check for the presence of a value in a sequence.
  - in, not in.
- Identity operators (is, is not) compare object identities.
  - is, is not.
- Bitwise operators (`&`, `|`, `^`, `~`, `<<`, `>>`) manipulate bits in binary representations.
  - `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), `~` (bitwise NOT), `<<` (left shift), `>>` (right shift).

### 5.3 Assignment, Comparison, & Identity Operators

- Understand pointers
  - Identity operators are used to compare the memory location of two objects
- Example 1
  - `x = [1, 2, 3]`
  - `y = x`
  - `print(x is y) # True`
- Example 2
  - `a = [1, 2, 3]`
  - `b = [1, 2, 3]`
  - `print(a is not b) # True`

### 5.4 Membership Operators

- The in operator returns True if a value is found in the sequence, otherwise False.
  - `my_list = [1, 2, 3, 4, 5]`
  - `print(3 in my_list) # True`
- The not in operator returns True if a value is not found in the sequence, otherwise False.
  - `my_tuple = ('apple', 'banana', 'orange')`
  - `print('grape' not in my_tuple) # True`

### 5.5 Operator Precedence

- PEMDAS, then Bitwise

### 6.2 Python Lists

- `my_list = [1, 2, 3, 'four', 5.0]`
- Key features:
  - Mutable: Elements can be modified, added, or removed
  - Ordered: Elements maintain their order
  - Indexing and Slicing: Access elements by index or extract sublists
  - Dynamic: Size can change during runtime
  - Common Operations: Append, extend, remove, and more

- Examples of Usage
  - Device Information Storage:
    - Use lists to store information about network devices, such as IP addresses, hostnames, and device types.
    - Code
      - `devices = [`
      - `{"hostname": "router1", "ip": "192.168.1.1", "type": "router"},`
      - `{"hostname": "switch1", "ip": "192.168.1.2", "type": "switch"},`
      - `# ...`
      - `]`
  - Configuration Templates:
    - Maintain lists for configuration templates that can be applied to multiple devices.
    - Code
      - `interface_configs = [`
      - `"interface GigabitEthernet0/1\n",`
      - `" description Connected to Server\n",`
      - `" ip address 192.168.1.1 255.255.255.0\n"`
      - `]`
  - Automated Tasks:
    - Use lists to iterate over devices and perform automated tasks, such as collecting data or applying configurations.
    - Code
      - `for device in devices:`
      - `connect_to_device(device["ip"])`
      - `collect_device_info(device)`
      - `apply_config_template(device, interface_configs)`
  - Error Handling:
    - Maintain lists to track devices that encounter errors during automation tasks.
    - Code
      - `devices_with_errors = []`
      - `for device in devices:`
      - `try:`
      - `# Perform automation tasks`
      - `except Exception as e:`
      - `devices_with_errors.append(device)`
      - `log_error(e)`
  - Use lists to store the results of operations or commands executed on network devices.



- `command_results = []`
- `for device in devices:`
- `result = execute_command(device, "show interfaces")`
- `command_results.append({"device": device["hostname"], "result": result})`
- Lists in network automation help structure and manage data efficiently, allowing for easy iteration, processing, and manipulation of information related to network devices.
- List Constructor
  - In Python, a list constructor is a way to create a new list. It's often used when you want to initialize a list with predefined values or when you want to convert an iterable (e.g., a string, tuple, or another list) into a list.
  - The `list()` constructor takes an iterable as an argument and returns a new list containing the elements of that iterable.
  - Example: Using List Constructor in Network Automation:
    - Suppose you have a list of device hostnames in a comma-separated string, and you want to convert it into a list for further processing.
    - 
    - `# Example: Using list constructor in network automation`
    - `device_names_string =`
    - `"router1,switch1,firewall1,router2,switch2"`
    - 
    - `# Convert comma-separated string to a list`
    - `device_names_list = list(device_names_string.split(','))`
    - 
    - `# Resulting list`
    - `print(device_names_list)`

### 6.3 Accessing List Items

- In Python lists, item indexes represent the position of an element in the list. Indexing is 0-based, meaning the first element has index 0, the second has index 1, and so on. Negative indexes count from the end, with -1 referring to the last element. Access elements using square brackets `[]`.
- Example:
  - `my_list = ['apple', 'banana', 'orange']`
  - 
  - `first_item = my_list[0]`     `# 'apple' (first element)`
  - `second_item = my_list[1]`    `# 'banana' (second element)`
  - `last_item = my_list[-1]`     `# 'orange' (last element)`
- Practical Example
  - `# List of network devices`

- devices = [
  - {"hostname": "router1", "ip": "192.168.1.1", "type": "router"},
  - {"hostname": "switch1", "ip": "192.168.1.2", "type": "switch"},
  - {"hostname": "firewall1", "ip": "192.168.1.3", "type": "firewall"},
  - # ... (more devices)
  - ]
  - 
  - # Retrieve information about the second device (index 1)
  - second\_device = devices[1]
  - 
  - # Access specific details using indexing
  - hostname = second\_device["hostname"]
  - ip\_address = second\_device["ip"]
  - device\_type = second\_device["type"]
  - 
  - # Display the information
  - print(f'Device: {hostname}, IP: {ip\_address}, Type: {device\_type}')
- Negative Indexing
  - Negative indexing in Python allows you to access elements from the end of a sequence, such as a list. The index -1 corresponds to the last element, -2 to the second-to-last, and so on. It provides a convenient way to access elements without needing to know the length of the sequence.
  - Example: Negative Indexing in Python Lists
    - network\_devices = ['router', 'switch', 'firewall', 'server', 'printer']
    - 
    - last\_device = network\_devices[-1] # 'printer' (last element)
    - second\_last\_device = network\_devices[-2] # 'server' (second-to-last element)
  - Practical Example
    - commands = ['show interfaces', 'show ip route', 'config terminal', 'interface GigabitEthernet0/1', 'shutdown']
    - 
    - # Retrieve the last command using negative indexing
    - last\_command = commands[-1]
    - 
    - # Display the last command
    - print(f'Last Command: {last\_command}')
    -
- Range of Indices

- In Python, the concept of range of indexes, also known as slicing, is used to access a subset of elements from a list, tuple, or string. The syntax is `sequence[start:stop:step]`, where:
  - `start` is the index where the slice starts (inclusive).
  - `stop` is the index where the slice ends (exclusive).
  - `step` is the interval between each index for slicing.
- default values:
  - `start=0`, `stop=len(sequence)`, `step=1`.
- Example
  - `ip_addresses = ["192.168.1.1", "192.168.1.2", "192.168.1.3", "192.168.1.4", "192.168.1.5"]`
  - `# Get the first three IP addresses`
  - `first_three_ips = ip_addresses[0:3]`
  - `print(first_three_ips) # Output: ['192.168.1.1', '192.168.1.2', '192.168.1.3']`
- Negative Indices
  - Use `-1`, `-2`, `-3`, ... in the first parameter followed by a colon to start from the back
  - Example
    - `ip_addresses = ["192.168.1.1", "192.168.1.2", "192.168.1.3", "192.168.1.4", "192.168.1.5"]`
    - `# Get the last three IP addresses`
    - **`last_three_ips = ip_addresses[-3:]`**
    - `print(last_three_ips) # Output: ['192.168.1.3', '192.168.1.4', '192.168.1.5']`
- Searching a List
  - In Python, the `in` keyword is used to check if a specified item is present in a list, tuple, string, or other iterable object. If the item is found, it returns `True`; otherwise, it returns `False`.
  - Example:
    - `ip_addresses = ["192.168.1.1", "192.168.1.2", "192.168.1.3", "192.168.1.4", "192.168.1.5"]`
    - `search_ip = "192.168.1.3"`
    - `# Check if the IP address is in the list`
    - `ip_present = search_ip in ip_addresses`
    - `print(ip_present) # Output: True`

## 6.4 Modifying Lists

- Changing List Values
  - In Python, the value of a specific item in a list can be changed by referring to the index of the item. The syntax is `list[index] = new_value`.
  - Example:
    - Here's an example where the value of a specific item in a list of device names is changed:

- `device_names = ["Switch1", "Router2", "Firewall3"]`
  - `# Change the name of the second device`
  - `device_names[1] = "Router_A"`
  - `print(device_names)` # Output: ['Switch1', 'Router\_A', 'Firewall3']
- Change a Range of Values in a List
  - `list[start:stop] = new_values`, where `new_values` is a list of new values
  - Example
    - `device_statuses = ["up", "down", "up", "up", "down"]`
    - `# Change the status of the second and third devices`
    - `device_statuses[1:3] = ["up", "up"]`
    - `print(device_statuses)` # Output: ['up', 'up', 'up', 'up', 'down']
- Inserting Items into a List
  - Use `insert()` method to add an item into a specific position in a list
  - `list.insert(index, element)`
    - where `index` is the position in the list where the new element should be inserted.
  - Example
    - `device_names = ["Switch1", "Router2", "Firewall3"]`
    - `# Insert a new device at the second position`
    - `device_names.insert(1, "Switch2")`
    - `print(device_names)` # Output: ['Switch1', 'Switch2', 'Router2', 'Firewall3']
- Appending to a List
  - `append()` method is used to add an item to the end of a list.
  - The syntax is `list.append(element)`, where `element` is the item to be added.
  - Example
    - `device_names = ["Switch1", "Router2", "Firewall3"]`
    - `# Append a new device to the list`
    - `device_names.append("Switch4")`
    - `print(device_names)` # Output: ['Switch1', 'Router2', 'Firewall3', 'Switch4']
- Extend List
  - the `extend()` method is used to add multiple items to the end of a list.
  - The syntax is `list.extend(iterable)`, where `iterable` can be a list, tuple, string, or any iterable object.
  - Example
    - `device_names = ["Switch1", "Router2", "Firewall3"]`
    - `# Extend the list with new devices`
    - `new_devices = ["Switch4", "Router5"]`
    - `device_names.extend(new_devices)`
    - `print(device_names)` # Output: ['Switch1', 'Router2', 'Firewall3', 'Switch4', 'Router5']

- Removing List Items
  - remove() method
    - removes the **first occurrence only** of a specified item from a list.
  - The syntax is list.remove(element), where element is the item to be removed.
  - Example
    - device\_names = ["Switch1", "Router2", "Firewall3", "Switch4"]
    - # Remove a device from the list
    - device\_names.remove("Switch4")
    - print(device\_names) # Output: ['Switch1', 'Router2', 'Firewall3']
- Popping a Specified Index
  - the pop() method
    - removes an item at a specified position in a list and return it.
  - If no index is specified, it removes and returns the last item in the list.
  - The syntax is list.pop(index), where index is the position of the item to be removed.
  - Example
    - device\_names = ["Switch1", "Router2", "Firewall3", "Switch4"]
    - # Pop the last device from the list
    - popped\_device = device\_names.pop()
    - print(popped\_device) # Output: 'Switch4'
    - print(device\_names) # Output: ['Switch1', 'Router2', 'Firewall3']
- Deleting a Index or Range
  - the del keyword is used to delete objects.
  - In the context of lists, it can be used to remove an item at a specific position, or to delete a slice of a list.
  - The syntax is del list[index] or del list[start:stop].
  - Example
    - device\_names = ["Switch1", "Router2", "Firewall3", "Switch4"]
    - # Delete the second device from the list
    - del device\_names[1]
    - print(device\_names) # Output: ['Switch1', 'Firewall3', 'Switch4']
- Clearing a List
  - Syntax
    - list\_name.clear()
  - Example
    - device\_names = ["Switch1", "Router2", "Firewall3", "Switch4"]
    - # Clear the list of device names
    - device\_names.clear()
    - print(device\_names) # Output: []

## 6.5 Sorting Lists

- Sort Ascending
  - the `sort()` method is used to sort the items in a list in ascending order. The syntax is `list.sort()`.
  - An optional key parameter can be provided to specify a function of one argument that is used to extract a comparison key from each element.
  - Example
    - `ip_addresses = ["192.168.1.3", "192.168.1.1", "192.168.1.2"]`
    - `# Sort the IP addresses`
    - `ip_addresses.sort(key=lambda ip: tuple(map(int, ip.split('.'))))`
    - `print(ip_addresses) # Output: ['192.168.1.1', '192.168.1.2', '192.168.1.3']`
- Sort Descending
  - In Python, the `reverse=True` keyword argument is used with the `sort()` method or the `sorted()` function to sort the items in a list in descending order.
  - By default, `sort()` and `sorted()` arrange elements in ascending order.
  - When `reverse=True` is provided, the order is reversed.
  - Example
    - `mac_addresses = ["00:0a:95:9d:68:16", "00:0a:95:9d:68:10", "00:0a:95:9d:68:15"]`
    - `# Sort the MAC addresses in descending order`
    - `mac_addresses.sort(reverse=True)`
    - `print(mac_addresses) # Output: ['00:0a:95:9d:68:16', '00:0a:95:9d:68:15', '00:0a:95:9d:68:10']`
- Case-Insensitive Sorts
  - a case-insensitive sort can be performed by using the `sort()` method or the `sorted()` function with a key parameter.
  - The key parameter is set to a function that converts each item to lowercase before comparison.
    - The built-in `str.lower` function can be used for this purpose.
  - Example
    - `device_names = ["Switch1", "router2", "Firewall3", "switch4"]`
    - `# Perform a case-insensitive sort`
    - `device_names.sort(key=str.lower)`
    - `print(device_names) # Output: ['Firewall3', 'router2', 'Switch1', 'switch4']`
- Reversing the Order of a List
  - the `reverse()` method is used to reverse the order of items in a list.
  - The syntax is `list.reverse()`.
    - This method modifies the original list and does not return any value.
  - Example
    - `device_names = ["Switch1", "Router2", "Firewall3", "Switch4"]`
    - `# Reverse the order of device names`

- `device_names.reverse()`
- `print(device_names)` # Output: ['Switch4', 'Firewall3', 'Router2', 'Switch1']
- Custom Sorts
  - the `sort()` method and the `sorted()` function can be customized using the `key` parameter.
  - The `key` parameter expects a function that defines the sorting criteria.
  - This function is applied to each element in the list, and the elements are sorted based on the values returned by this function.
  - Example
    - `devices = [("Switch1", 20), ("Router2", 15), ("Firewall3", 30), ("Switch4", 25)]`
    - `# Sort the devices based on their uptime`
    - `devices.sort(key=lambda device: device[1])`
    - `print(devices)` # Output: [('Router2', 15), ('Switch1', 20), ('Switch4', 25), ('Firewall3', 30)]

## 6.6 Copying a List

- Using the `copy()` method: This method creates a new list by copying the original list.
  - `original_list = [1, 2, 3, 4, 5]`
  - `copied_list = original_list.copy()`
- Using the `list()` constructor: This function creates a new list from the original list.
  - `original_list = [1, 2, 3, 4, 5]`
  - `copied_list = list(original_list)`
- Example of Creating a backup
  - `network_devices = ['Switch1', 'Router2', 'Firewall3']`
  - `backup_devices = network_devices.copy()`
  - `# or`
  - `backup_devices = list(network_devices)`

## 6.7 Concatenating Lists

- lists can be concatenated using the `+` operator or the `extend()` method.
- Example of `+`
  - `list1 = [1, 2, 3]`
  - `list2 = [4, 5, 6]`
  - `concatenated_list = list1 + list2` # The result is [1, 2, 3, 4, 5, 6]
- Example of `.extend()`
  - `list1 = [1, 2, 3]`
  - `list2 = [4, 5, 6]`
  - `list1.extend(list2)` # list1 is now [1, 2, 3, 4, 5, 6]

## 6.8 Python Tuple

- immutable sequences of arbitrary elements. They are defined by enclosing elements in parentheses `()`, separated by commas. For example, `my_tuple = (1, "a", 3.14)`.

- Smaller memory footprint, great for LARGE DATASETS
- Example Format
  - # Define a tuple for a network device
  - network\_device = ("192.168.1.1", "cisco", "admin", "password")
  - 
  - # Unpack the tuple
  - ip\_address, device\_type, username, password = network\_device
  - 
  - # Use the unpacked values
  - print(f'Connecting to device {device\_type} at {ip\_address} using username {username}')
- Tuple Length
  - # Creating a tuple
  - network\_devices = ('Router1', 'Switch1', 'Firewall1', 'Server1')
  - 
  - # Determining the length of the tuple
  - num\_devices = len(network\_devices)
  - 
  - # Printing the result
  - print(f'The number of network devices is: {num\_devices}')
- Tuples with Single Item
  - A tuple with one item is created by placing a single value inside parentheses () and following it with a comma.
  - Example
    - # Define a tuple with a single item
    - username = ("admin",)
    - 
    - # Unpack the tuple
    - (username,) = username
    - 
    - # Use the unpacked value
    - print(f'Username: {username}')
    -
- The Tuple Constructor
  - The tuple() constructor allows the creation of a tuple.
  - It can take an iterable as an argument and convert it into a tuple.
  - This can be useful when there is a need to ensure the immutability of a sequence of data.
  - Example
    - # Define a list with network device information



- network\_device\_info = ["192.168.1.1", "cisco", "admin", "password"]
- 
- # Use the tuple() constructor to convert the list into a tuple
- network\_device = tuple(network\_device\_info)
- 
- # Print the tuple
- print(f'Network device: {network\_device}')
- Modifying Tuples
  - Accessing Items in Tuples
    - # Define a tuple for a network device
    - network\_device = ("192.168.1.1", "cisco", "admin", "password")
    - 
    - # Access the IP address (first item in the tuple)
    - ip\_address = network\_device[0]
    - 
    - # Access the device type (second item in the tuple)
    - device\_type = network\_device[1]
    - 
    - # Print the accessed items
    - print(f'IP Address: {ip\_address}, Device Type: {device\_type}')
- Range of Indexes
  - Slicing in Python is done by specifying a start index and an end index separated by a colon : inside square brackets [].
    - The start index is inclusive, and the end index is exclusive.
    - If the start index is omitted, slicing starts from the beginning of the tuple.
    - If the end index is omitted, slicing goes until the end of the tuple.
  - Example:
    - # Define a tuple for a network device
    - network\_device = ("192.168.1.1", "cisco", "admin", "password", "port 22", "ssh")
    - 
    - # Access a range of items in the tuple (from the third item to the fifth item)
    - credentials = network\_device[2:5]
    - 
    - # Print the accessed items
    - print(f'Credentials: {credentials}')
- Range of Negative Indexes
  - The last item in a tuple is at index -1, the second last item is at index -2
  - To specify a range of negative indexes in a tuple, slicing can be used in the same way as with positive indexes.

- The start index is inclusive, and the end index is exclusive.
- Example
  - # Define a tuple for a network device
  - network\_device = ("192.168.1.1", "cisco", "admin", "password", "port 22", "ssh")
  - 
  - # Access a range of items in the tuple (from the third last item to the last item)
  - connection\_info = network\_device[-3:]
  - 
  - # Print the accessed items
  - print(f'Connection Info: {connection\_info}')
- Checking an Item in a Tuple
  - To determine whether a specific item exists in a tuple, the in keyword can be used
  - This keyword checks if a value is found in a sequence like a tuple.
  - If the value is in the tuple, the expression returns True. Otherwise, it returns False.
  - Example
    - # Define a tuple for a network device
    - network\_device = ("192.168.1.1", "cisco", "admin", "password")
    - 
    - # Check if 'cisco' is in the tuple
    - device\_exists = 'cisco' in network\_device
    - 
    - # Print the result
    - print(f'Device exists: {device\_exists}')

## 6.10 Updating Tuples

- Tuples in Python are immutable, meaning their values cannot be changed once they are created. However, there is a workaround to change the values in a tuple.
- This involves:
  - converting the tuple into a list,
  - changing the list
  - converting the list back into a tuple.
- Example
  - # Define a tuple for a network device
  - network\_device = ("192.168.1.1", "cisco", "admin", "password")
  - 
  - # Convert the tuple into a list
  - network\_device\_list = list(network\_device)
  - 
  - # Change the IP address in the list

- network\_device\_list[0] = "192.168.2.2"
- 
- # Convert the list back into a tuple
- network\_device = tuple(network\_device\_list)
- 
- # Print the new tuple
- print(f'Network device: {network\_device}')
- Adding Items to a Tuple
  - Items cannot be added directly to a tuple due to its immutable nature.
  - However, there is a workaround to add items to a tuple.
  - This involves concatenating the tuple with another tuple containing the new items.
  - Example:
    - # Define a tuple for a network device
    - network\_device = ("192.168.1.1", "cisco", "admin", "password")
    - 
    - # Define a new tuple with additional information
    - additional\_info = ("port 22", "ssh")
    - 
    - # Concatenate the tuples
    - network\_device = network\_device + additional\_info
    - 
    - # Print the new tuple
    - print(f'Network device: {network\_device}')
- Removing Items from a Tuple
  - Steps
    - Convert the tuple to a list
    - remove the item
    - convert the list back to a tuple.
  - Example
    - # Original tuple with IP address, hostname, and model
    - device\_info = ("192.168.1.1", "switch1", "Cisco 3750")
    - 
    - # Convert tuple to list
    - device\_list = list(device\_info)
    - 
    - # Remove old model from list
    - device\_list.remove("Cisco 3750")
    - 
    - # Add new model to list
    - device\_list.append("Cisco 3850")

- 
- # Convert list back to tuple
- device\_info = tuple(device\_list)
- Other
  - Shallow Copy
    - copy()
    - Changes to the copy do not affect the original
  - Deep Copy
    - Changes to the copy do affect the original

## 7.2 Python Dictionaries

- Unordered, Mutable collection of key-value pairs
- Purpose
  - Key-Value Mapping
    - Dictionaries allow you to associate values with descriptive keys, providing a convenient way to represent relationships between data.
  - Fast Retrieval
    - Due to hashing, dictionaries offer fast and efficient retrieval of values based on keys.
  - Flexibility
    - Dictionaries are versatile for representing complex data structures, configurations, and mappings.
- Example
  - # Dictionary representing network devices
  - network\_devices = {
  - 'Router1': '192.168.1.1',
  - 'Switch1': '10.0.0.1',
  - 'Firewall1': '172.16.0.1',
  - 'Server1': '192.168.2.1',
  - }
  - 
  - # Accessing information about a specific device
  - router\_ip = network\_devices['Router1']
  - 
  - # Adding a new device to the dictionary
  - network\_devices['Switch2'] = '10.0.0.2'
  - 
  - # Resulting updated dictionary
  - # {'Router1': '192.168.1.1', 'Switch1': '10.0.0.1', 'Firewall1': '172.16.0.1', 'Server1': '192.168.2.1', 'Switch2': '10.0.0.2'}
- Creating Dictionaries

- In Python, you can create dictionaries using curly braces {} and specifying key-value pairs separated by colons
- Example:
  - # Creating a dictionary with key-value pairs
  - network\_devices = {
    - 'Router1': '192.168.1.1',
    - 'Switch1': '10.0.0.1',
    - 'Firewall1': '172.16.0.1',
    - 'Server1': '192.168.2.1',
    - }
- Dictionary of Dictionaries
  - # Creating a dictionary with device information
  - device\_info = {
    - 'Router1': {'ip': '192.168.1.1', 'vendor': 'Cisco', 'os': 'IOS'},
    - 'Switch1': {'ip': '10.0.0.1', 'vendor': 'Cisco', 'os': 'IOS-XE'},
    - 'Firewall1': {'ip': '172.16.0.1', 'vendor': 'PaloAlto', 'os': 'PAN-OS'},
    - }
- Dictionary Data Types
  - In Python dictionaries, the keys and values can be of various data types. Commonly supported data types include:
    - Keys
      - Keys in dictionaries can be of **any immutable data type**, such as integers, strings, or tuples.
      - Lists and dictionaries, which are mutable, cannot be used as keys.
    - Values
      - Values can be of **any data type**, including integers, floats, strings, lists, dictionaries, tuples, and more.
      - The flexibility of Python dictionaries allows for the storage of diverse and nested data structures.
  - Example:
    - # Dictionary with various data types
    - mixed\_data\_types = {
      - 'integer\_key': 42,
      - 'string\_key': 'hello',
      - 'list\_key': [1, 2, 3],
      - 'nested\_dict\_key': {'nested\_key': 'value'},
      - 'tuple\_key': (4, 5, 6),
      - }
- Python Dictionary Constructor

- The Python dictionary constructor, dict(), is a built-in function that allows you to create a dictionary from various data structures. It can take no arguments, a dictionary, or an iterable containing key-value pairs.
- Role and Function of the Dictionary Constructor
  - Creating Empty Dictionaries:
    - You can use dict() to create an empty dictionary.
  - Converting from Iterables:
    - It can convert a list of tuples or other iterable sequences into a dictionary.
  - Copying Dictionaries:
    - It can create a new dictionary by copying the contents of an existing dictionary.
- Example:
  - # Example: Creating a dictionary using the constructor for network devices
  - device\_list = [('Router1', '192.168.1.1'), ('Switch1', '10.0.0.1'), ('Firewall1', '172.16.0.1')]
  - 
  - # Using the dictionary constructor to convert a list of tuples into a dictionary
  - network\_devices = dict(device\_list)
  - 
  - # Resulting dictionary: {'Router1': '192.168.1.1', 'Switch1': '10.0.0.1', 'Firewall1': '172.16.0.1'}
- Access Dictionary Values
  - In Python, items in a dictionary can be accessed using their keys. Here is a general example:
  - Example
    - # Define a dictionary
    - dict1 = {"key1": "value1", "key2": "value2", "key3": "value3"}
    - 
    - # Access an item using its key
    - value1 = dict1["key1"]
  - In the context of network automation, dictionaries might be used to store information about network devices. For example, a dictionary might contain the IP address, hostname, and model of a network device.
  - Example:
    - # Define a dictionary with device information
    - device\_info = {"ip\_address": "192.168.1.1", "hostname": "switch1", "model": "Cisco 3750"}

- 
- # Access an item using its key
- ip\_address = device\_info["ip\_address"]
- hostname = device\_info["hostname"]
- model = device\_info["model"]

### 7.3 Modifying Dictionary Items

- Changing Values in a Python Dictionary:
  - # Creating a dictionary
  - network\_devices = {
  - 'Router1': '192.168.1.1',
  - 'Switch1': '10.0.0.1',
  - 'Firewall1': '172.16.0.1',
  - 'Server1': '192.168.2.1',
  - }
  - 
  - # Changing the value of a specific key
  - network\_devices['Router1'] = '192.168.1.100'
- Changing Values in a Dictionary within a Dictionary
  - # Device information dictionary
  - device\_info = {
  - 'Router1': {'ip': '192.168.1.1', 'vendor': 'Cisco', 'os': 'IOS'},
  - 'Switch1': {'ip': '10.0.0.1', 'vendor': 'Cisco', 'os': 'IOS-XE'},
  - 'Firewall1': {'ip': '172.16.0.1', 'vendor': 'PaloAlto', 'os': 'PAN-OS'},
  - }
  - 
  - # Changing the IP address of a specific router
  - device\_info['Router1']['ip'] = '192.168.1.100'
- Removing an Item from a Dictionary
  - To remove an item from a Python dictionary, the del keyword or the pop() method can be used.
  - del example
    - dictionary = {'key1': 'value1', 'key2': 'value2'}
    - del dictionary['key1']
  - pop example
    - dictionary = {'key1': 'value1', 'key2': 'value2'}
    - value = dictionary.pop('key1')
- Updating a Python Dictionary
  - In Python, the update() method is used to update a dictionary with the key/value pairs from another dictionary, or from an iterable of key/value pairs.
  - # Define two dictionaries

- dict1 = {"key1": "value1", "key2": "value2"}
- dict2 = {"key2": "new\_value2", "key3": "value3"}
- 
- # Update dict1 with dict2
- dict1.update(dict2)
- 
- 
- After this code is executed, dict1 will be {"key1": "value1", "key2": "new\_value2", "key3": "value3"}.
- In the context of network automation, the update() method might be used to update information about a network device.
  - # Define a dictionary with device information
  - device\_info = {"ip\_address": "192.168.1.1", "hostname": "switch1", "model": "Cisco 3750"}
  - 
  - # Define a dictionary with updated device information
  - updated\_info = {"model": "Cisco 3850", "os\_version": "15.2"}
  - 
  - # Update device\_info with updated\_info
  - device\_info.update(updated\_info)
- Adding Dictionary Items
  - To add an item to a Python dictionary, simply assign a value to a new key in the dictionary. Here is an example:
    - dictionary = {'key1': 'value1', 'key2': 'value2'}
    - dictionary['key3'] = 'value3'
- Clearing a Dictionary
  - To clear a Python dictionary, the clear() method is used. Here is an example:
    - dictionary = {'key1': 'value1', 'key2': 'value2'}
    - dictionary.clear()

## 7.4 Python Sets

- unordered, mutable collection of unique items
- use a set when the order does not matter and simple storage is needed
- allow of efficient membership tests, meaning it is easier to check whether an item exists compared to other data types
- Example:
  - network\_devices = {'Switch', 'Router', 'Firewall'}
  - new\_devices = {'Firewall', 'Load Balancer'}
  - 
  - # Add new devices to the network
  - network\_devices = network\_devices.union(new\_devices)



- In this case, a 'Load Balancer' is added to the network\_devices set.
- Creating a Set
- A Python set is created by placing a comma-separated sequence of items inside curly braces {}. Alternatively, the set() function can be used to create a set from a list or other iterable. Here is an example:
  - # Using curly braces
  - set1 = {'item1', 'item2', 'item3'}
  - 
  - # Using the set() function
  - set2 = set(['item1', 'item2', 'item3'])
- Determine the Number of Items in a Set
  - The number of items in a Python set can be determined using the len() function. Here is an example:
    - set1 = {'item1', 'item2', 'item3'}
    - number\_of\_items = len(set1)
- The set() Constructor
  - The Python set constructor, set(), is a built-in function for creating a set. It takes an iterable (like a list or a string) as an argument and returns a set containing the unique elements of the iterable. If no argument is given, it creates an empty set.
  - Here is an example:
    - # Using a list as an argument
    - set1 = set(['item1', 'item2', 'item3', 'item1'])
    - 
    - # Using a string as an argument
    - set2 = set('hello')
- Accessing Items in a Set
  - Items in a Python set cannot be accessed by referring to an index or a key, because sets are unordered collections of items. However, one can loop through the set items using a for loop, or ask if a specified value is present in a set by using the in keyword.
    - Here is an example of looping through a set:
      - 
      - set1 = {'item1', 'item2', 'item3'}
      - for item in set1:
      - print(item)

## 7.5 Modifying Python Sets

- Add Items to a Set
  - To add an item to a Python set, the add() method is used. Here is an example
    - set1 = {'item1', 'item2', 'item3'}
    - set1.add('item4')
- Adding Sets

- To add items from another set into the current set in Python, the `update()` method is used. This method takes one or more sets as arguments and adds all their elements to the current set. Here is an example:
  - `set1 = {'item1', 'item2', 'item3'}`
  - `set2 = {'item4', 'item5'}`
  - `set1.update(set2)`
- Removing an Item from a Set
  - To remove an item from a Python set, the `remove()` or `discard()` method can be used. The `remove()` method removes the specified item from the set, but if the item does not exist, it raises an error. The `discard()` method also removes the specified item from the set, but it does not raise an error if the item does not exist.
    - `set1 = {'item1', 'item2', 'item3'}`
    - `set1.remove('item1')`
- Emptying a Set
  - To empty a Python set, the `clear()` method is used. Here is an example:
    - `set1 = {'item1', 'item2', 'item3'}`
    - `set1.clear()`
- Deleting a Set
  - In Python, you can delete a set using the `del` keyword. This removes the entire set from memory, and attempting to use the set afterward will result in a `NameError` because the set no longer exists.
    - `# Creating a set`
    - `network_devices = {'Router1', 'Switch1', 'Firewall1'}`
    - `# Using del to delete the set`
    - `del network_devices`

## 8.2 Automating Tasks with Python

- Configuration Management
  - Configuration management can be automated using Python by leveraging libraries and frameworks that facilitate interactions with network devices, servers, or infrastructure components. One popular tool for network automation and configuration management is Netmiko, a multi-vendor library for managing network devices.
  - 
  - Example:
    - Here's an example using Python and Netmiko to automate the configuration of a network device. In this case, let's consider configuring an interface on a Cisco router:
      - `from netmiko import ConnectHandler`
      - 
      - `# Define the device parameters`

- device = {
  - 'device\_type': 'cisco\_ios',
  - 'ip': '192.168.1.1',
  - 'username': 'admin',
  - 'password': 'password',
  - 'secret': 'enable\_password',
  - }
  - 
  - # Connect to the device
  - net\_connect = ConnectHandler(\*\*device)
  - net\_connect.enable()
  - 
  - # Define the configuration commands
  - interface\_config = [
    - 'interface GigabitEthernet0/0',
    - 'ip address 192.168.2.1 255.255.255.0',
    - 'no shutdown',
    - 'exit',
  - ]
  - 
  - # Send configuration commands to the device
  - output = net\_connect.send\_config\_set(interface\_config)
  - 
  - # Display the output
  - print(output)
  - 
  - # Disconnect from the device
  - net\_connect.disconnect()
- Task Scheduling
  - Task scheduling in Python can be automated using the schedule library, which provides a simple interface for scheduling and running periodic tasks. Below is an example that demonstrates how to use the schedule library to schedule a Python function to run at specific intervals:
  - First, you need to install the schedule library if you haven't already:
    - pip install schedule
  - Now, you can create a Python script with an example of task scheduling:
    - import schedule
    - import time
    - 
    - def my\_task():

- `print("Executing my_task at", time.strftime("%Y-%m-%d %H:%M:%S"))`
- 
- `# Schedule the task to run every 1 minute`
- `schedule.every(1).minutes.do(my_task)`
- 
- `# Alternatively, you can schedule the task using a cron-like syntax`
- `# schedule.every().hour.at(":30").do(my_task)`
- 
- `# Run the scheduler continuously`
- `while True:`
- `schedule.run_pending()`
- `time.sleep(1)`
- Cloud Automation
  - Cloud automation using Python can be achieved through the use of cloud provider SDKs (Software Development Kits) or APIs (Application Programming Interfaces). Each major cloud provider (e.g., AWS, Azure, Google Cloud) offers SDKs that allow developers to interact with and automate cloud resources. Below is an example using the Boto3 library, which is the official Python SDK for Amazon Web Services (AWS).
  - First, you need to install the Boto3 library if you haven't already:
    - `pip install boto3`
  - Now, you can create a Python script to automate a simple AWS task, such as creating an S3 bucket:
    - `import boto3`
    - 
    - `# AWS credentials (replace with your own credentials)`
    - `aws_access_key = 'YOUR_ACCESS_KEY'`
    - `aws_secret_key = 'YOUR_SECRET_KEY'`
    - `region_name = 'us-east-1'`
    - 
    - `# Create an S3 client`
    - `s3 = boto3.client('s3', aws_access_key_id=aws_access_key,`  
`aws_secret_access_key=aws_secret_key, region_name=region_name)`
    - 
    - `# Specify the bucket name`
    - `bucket_name = 'my-unique-bucket-name'`
    - 
    - `# Create an S3 bucket`
    - `try:`

- response = s3.create\_bucket(Bucket=bucket\_name)
- print(f'Bucket '{bucket\_name}' created successfully.")
- except Exception as e:
- print(f'Error creating bucket: {e}")
- In this example:
  - Replace YOUR\_ACCESS\_KEY and YOUR\_SECRET\_KEY with your AWS access key and secret key.
  - The script uses the Boto3 library to create an S3 client with the specified credentials and region.
  - It specifies a unique bucket name and attempts to create an S3 bucket using s3.create\_bucket().
- 
- Create a Function
  - A Python function is defined using the def keyword, followed by a function name, parentheses (), and a colon :. The function body is indented and includes any number of statements to be executed when the function is called. Here is an example:
    - def function\_name():
    - # function body
    - pass
- Calling a Function
  - A Python function is called by using its name followed by parentheses (). If the function takes arguments, the arguments are placed inside the parentheses. Here is an example:
    - def function\_name():
    - # function body
    - pass
    - 
    - # Call the function
    - function\_name()
- Arguments
  - Arguments in Python are values that are passed to a function when it is called. Each argument is treated as a variable inside the function. Arguments are used to provide inputs to a function so it can perform a task based on those inputs.
    - def add\_device(network\_devices, device):
    - network\_devices.add(device)
    - return network\_devices
- Keyword Arguments
  - In this case, keyword1 and keyword2 are keyword arguments with default values value1 and value2.

- `def function_name(keyword1=def_value1, keyword2=def_value2):`
- `# function body`
- `pass`
- `configure_device(device_type='Router', ip_address='192.168.1.2')`
- `# This last line will update the keyword arguments`
- Arbitrary Keyword Arguments
  - Arbitrary keyword arguments in Python allow a function to accept any number of keyword arguments. This is useful when the exact number of keyword arguments is not known in advance. Arbitrary keyword arguments are defined by prefixing the argument name with double asterisks `**` in the function definition.
  - Here is an example of a function with arbitrary keyword arguments:
    - `def function_name(**kwargs):`
    - `for key, value in kwargs.items():`
    - `print(f'{key}: {value}')`
- Default Parameter Values
  - A default parameter value in Python is a value that is assigned to a function parameter when the function is defined. If the function is called without an argument for that parameter, the default value is used. Default parameter values are specified by using the assignment operator `=` in the function definition.
    - Here is an example of a function with a default parameter value:
      - `def function_name(parameter='default value'):`
      - `# function body`
      - `pass`
    - In this case, if `function_name` is called without an argument, `parameter` will be `'default value'`.
  - Consider a function that configures a network device with a default IP address:
    - `def configure_device(device, ip_address='192.168.1.1'):`
    - `# function body`
    - `pass`
- Passing a List as an Argument
  - A list can be passed as an argument to a Python function just like any other data type. The function can then perform operations on the list. Here is an example:
    - `def function_name(list_parameter):`
    - `for item in list_parameter:`
    - `print(item)`
  - In this case, `list_parameter` is a list that is passed to `function_name`. The function prints each item in the list.
  - Consider a function that configures a list of network devices:
    - `def configure_devices(device_list):`
    - `for device in device_list:`

- `print(f'Configuring {device}')`
  - `devices = ['Switch', 'Router', 'Firewall']`
  - `configure_devices(devices)`
- Returning a Value
  - A Python function returns a value using the return statement. The value that follows the return keyword is the result that the function sends back when it is called. If no value is specified, the function will return None.
  - Here is an example of a function that returns a value:
    - `def function_name():`
    - `return 'value'`
- Function Recursion
  - Function recursion in Python is a process in which a function calls itself as a subroutine. This allows the function to be repeated several times, as it can call itself during its execution. Recursion can be direct or indirect. A direct recursion occurs if a function calls itself; an indirect recursion involves the function calling another function, which eventually results in the original function being called.
  - Recursion can be a powerful tool, but it can also be computationally expensive and cause a program to crash if not implemented with care. It is important to ensure that a recursion has a base case that will be met, which will stop the recursion.
  - Here is an example of a recursive function:
    - `def countdown(n):`
    - `if n <= 0:`
    - `print('Liftoff!')`
    - `else:`
    - `print(n)`
    - `countdown(n-1)`
  - In this case, countdown is a function that takes an argument n. If n is less than or equal to 0, it prints 'Liftoff!'. Otherwise, it prints n and then calls itself with the argument n-1.
- \*argument
  - Use this to accept any number of arguments for a function, including 0
  - Example
    - `def add_devices(network_devices, *devices):`
    - `for device in devices:`
    - `network_devices.add(device)`
    - `return network_devices`

## 9.2 If/Else Statements

- Purpose: Used for conditional execution of code based on whether a specified condition is true or false.

- Syntax:
  - if condition:
    - # code to execute if the condition is true
  - else:
    - # code to execute if the condition is false
- Flow: The program checks the specified condition. If it's true, the code inside the if block is executed; otherwise, the code inside the else block is executed.

### 9.3 Creating If Statements

- Conditions
  - Expressions that evaluate to either True or False.
    - `x == y` (Is x equal to y?)
    - `x < y` (Is x less than y?)
    - `x > y` (Is x greater than y?)
    - `x != y` (Is x not equal to y?)
- elif
  - `x = 20`
  - 
  - if `x < 10`:
    - `print("x is less than 10")`
  - elif `x < 30`:
    - `print("x is less than 30 but not less than 10")`
  - else:
    - `print("x is 30 or more")`
- else
  - `x = 20`
  - 
  - if `x < 10`:
    - `print("x is less than 10")`
  - else:
    - `print("x is 10 or more")`
  -

### 9.4 Conditional Expressions

- and Logical Operator
  - The and keyword in Python is a logical operator used in conditional statements. It returns True if both the conditions on its left and right are True. If either or both conditions are False, it returns False.
    - `ip_address = "192.168.1.1"`
    - `first_octet = int(ip_address.split(".")[0])`
    - 
    - if `first_octet >= 1` and `first_octet <= 126`:



- `print(f'{ip_address} is a Class A IP address.')`
  - `else:`
  - `print(f'{ip_address} is not a Class A IP address.')`
- or Logical Operator
  - The or keyword in Python is a logical operator used in conditional statements. It returns True if either or both of the conditions on its left and right are True. If both conditions are False, it returns False.
    - `ip_address = "192.168.1.1"`
    - `first_octet = int(ip_address.split(".")[0])`
    - 
    - `if first_octet == 10 or first_octet == 172 or first_octet == 192:`
    - `print(f'{ip_address} is a private IP address.')`
    - `else:`
    - `print(f'{ip_address} is not a private IP address.')`
- and versus or
  - In Python, and and or are logical operators used in conditional statements. Here's how they differ:
  - The and operator returns True if both the conditions on its left and right are True. If either or both conditions are False, it returns False.
  - The or operator returns True if either or both of the conditions on its left and right are True. If both conditions are False, it returns False.
    - `x = 10`
    - `y = 20`
    - 
    - `if x > 5 and y > 5:`
    - `print("Both x and y are greater than 5")`
    - `else:`
    - `print("Either x or y is not greater than 5")`
    - 
    - `if x > 5 or y < 5:`
    - `print("Either x is greater than 5 or y is less than 5")`
    - `else:`
    - `print("Neither x is greater than 5 nor y is less than 5")`
- not Logical Operator
  - The not keyword in Python is a logical operator used in conditional statements. It returns True if the condition following it is False, and False if the condition is True. In other words, it reverses the truth value of the condition.
    - `ip_address = "192.168.1.1"`
    - `first_octet = int(ip_address.split(".")[0])`
    -

- if not first\_octet == 127:
- print(f'{ip\_address} is not a loopback IP address.')
  - else:
  - print(f'{ip\_address} is a loopback IP address.')
    - print(f'Device {device\_ip} is {status}.')
- Shorthand if/else
  - A shorthand if statement, also known as a one-liner if statement or a ternary operator, is a way to write an if/else statement in a single line in Python. It's used when there are simple, short conditions to be checked and actions to be taken.
  - Here's the syntax of a shorthand if statement:
    - <value\_if\_true> if <condition> else <value\_if\_false>
  - This will return value\_if\_true if condition is True, and value\_if\_false if condition is False.
  - # Assume there is a function 'is\_device\_up(device\_ip)' that returns True if the device is up and False if it's down.
    - device\_ip = "192.168.1.1" # IP address of the network device
    - status = "up" if is\_device\_up(device\_ip) else "down"
    - print(f'Device {device\_ip} is {status}.')
- A function is needed to identify the network status. Create a function name "check\_network\_status()". The function should accept two Boolean values representing a network connection and firewall status.
  - If both the connection and firewall are True, return "No issues detected"
  - If the connection is True and the firewall is False, return "Proceed with caution"
  - If the connection is False, return "Network not detected"
  - For any other situation, return "Unexpected network status"
    - def check\_network\_status(bool1, bool2):
    - if type(bool1) == bool and type(bool2) == bool:
    - if bool1 == True and bool2 == True:
    - return "No issues detected"
    - elif bool1 == True:
    - return "Proceed with caution"
    - elif bool1 == False:
    - return "Network not detected"
    - else:
    - return "Unexpected network status"

## 10.2 Python Loops

- What Are Loops?
  - Python loops are control flow structures used to repeatedly execute a block of code. There are two types of loops in Python: for and while.
-

- A for loop is used to iterate over a sequence (like a list, tuple, dictionary, string, or range) or other iterable object.
- A while loop is used to repeatedly execute a block of code as long as a certain condition is True.
- Loops are used when there's a need to perform a task multiple times, such as processing items in a list one by one or running a block of code until a certain condition is met.
  - subnet = "192.168.1."
  - ip\_addresses = [subnet + str(i) for i in range(1, 256)]
  - 
  - for ip in ip\_addresses:
  - print(ip)
- Example Problem
  - Complete the function "generate\_users()" to generate sequentially numbered usernames starting at 1 until an indicated end value. The function should accept a string and a numeric value, generate usernames beginning with the string and ending with increasing numeric values (e.g., "test\_account1", "test\_account2", "test\_account3"), returning the usernames as a set to preserve uniqueness.
  - My Answer
    - def generate\_users(username\_string, num\_accounts):
    - count = 1
    - username\_set = set()
    - while count <= num\_accounts:
    - username\_set.add(f'{username\_string} {count}')
    - count+=1
    - return username\_set
    - 
    - print(generate\_users("test\_account", 4))
  - Expected Answer
    - user\_set = set()
    - for i in range(1, num\_accounts+1):
    - user\_set.add(f'{username\_string} {i}')
    - return user\_set
  - Most Optimal Answer
    - def generate\_users(username\_string, num\_accounts):
    - return {f'{username\_string} {i}' for i in range(1, num\_accounts + 1)}

### 10.3 Constructing While Loops

- A while loop in Python is a control flow structure that repeatedly executes a block of code as long as a certain condition is True.
  - subnet = "192.168.1."
  - ip\_addresses = []

- i = 1
- 
- while i < 256:
- ip\_addresses.append(subnet + str(i))
- i += 1
- 
- for ip in ip\_addresses:
- print(ip)
- Incrementing in a while Loop
  - The increment in a while loop is a step that increases a counter variable. It's necessary to prevent the loop from running indefinitely, ensuring that the loop condition eventually becomes False, thus terminating the loop. Here's an example:
    - i = 0
    - while i < 5:
    - print(i)
    - i += 1 # Increment
  - In this script, i += 1 is the increment. It increases the value of i by 1 in each iteration of the loop. Without this increment, the loop would run forever because i would always be less than 5, making the loop condition i < 5 always True.
- The break Statement
  - The break statement in Python is used to exit a loop prematurely. It's used when there's a need to stop the loop even if the loop's condition has not become False
    - subnet = "192.168.1."
    - ip\_addresses = []
    - i = 1
    - 
    - while True:
    - if i > 255:
    - break
    - ip\_addresses.append(subnet + str(i))
    - i += 1
    - 
    - for ip in ip\_addresses:
    - print(ip)
- continue Statement
  - The continue statement in Python is used in loops to skip the rest of the current iteration and move directly to the next one.
    - subnet = "192.168.1."
    - ip\_addresses = []

- i = 0
- 
- while i < 256:
- i += 1
- if i == 100: # Let's say we want to skip this IP
- continue
- ip\_addresses.append(subnet + str(i))
- 
- for ip in ip\_addresses:
- print(ip)
- else Statement
  - The else statement in a Python while loop specifies a block of code to be executed when the loop condition becomes False. The else block executes after the loop finishes, but not if the loop is exited prematurely with a break statement.
  - subnet = "192.168.1."
  - ip\_addresses = []
  - i = 0
  - 
  - while i < 256:
  - i += 1
  - ip\_addresses.append(subnet + str(i))
  - else:
  - print("All IP addresses have been generated.")
  - 
  - for ip in ip\_addresses:
  - print(ip)

#### 10.4 Constructing for Loops

- A for loop in Python is a control flow statement that allows code to be executed repeatedly. The structure of a for loop is as follows:
  - for variable in iterable:
  - # code to be executed
- Example:
  - subnet = "192.168.1."
  - ip\_addresses = [subnet + str(i) for i in range(1, 256)]
  - 
  - for ip in ip\_addresses:
  - print(ip)
- for loops versus while Loops
- For loops and while loops are both control flow statements in Python that allow code to be executed repeatedly, but they differ in their usage and control conditions:

for Loop: A for loop is used for iterating over a sequence (like a list, tuple, dictionary, set, or string) or other iterable object. The set of statements is executed once for each item in the list. The loop continues until it has gone through each item in the sequence.

- for variable in iterable:
  - # code to be executed
- while Loop: A while loop is used when a set of statements needs to be executed until a condition is false. The condition is checked before each iteration, and if it evaluates to true, the loop continues; if it evaluates to false, the loop ends.
  - while condition:
    - # code to be executed
- Looping Through a String
  - Looping through a string in Python is straightforward due to Python's built-in support for string iteration. Here's the basic syntax:
    - for character in string:
      - # code to be executed
  - Example:
    - ip\_address = "192.168.1.1"
    - octets = ip\_address.split(".")
    - octet\_count = 0
    - 
    - for octet in octets:
      - if octet.isdigit() and 0 <= int(octet) <= 255:
      - octet\_count += 1
      -
    - print(f"The IP address {ip\_address} has {octet\_count} octets.")
- break Statement
  - The break statement in Python is used to exit or "break" a for or while conditional loop. When the loop encounters the break statement, the control flow is immediately interrupted and the program proceeds to the next line of code outside the loop.
  - Example:
    - ip\_address = "192.168.1.1"
    - octets = ip\_address.split(".")
    - 
    - for octet in octets:
      - if not octet.isdigit() or not 0 <= int(octet) <= 255:
      - print(f"{ip\_address} is not a valid IP address.")
      - break
    - else:
    - print(f"{ip\_address} is a valid IP address.")

- continue Statement
  - The continue statement in Python is used within a loop (for or while) to skip the rest of the current iteration and move directly to the next one. When the continue statement is encountered, the program control jumps to the top of the loop, and the next iteration begins.
  - Example:
    - ip\_address = "192.168.1.1"
    - octets = ip\_address.split(".")
    - 
    - for octet in octets:
    - if not octet.isdigit() or not 0 <= int(octet) <= 255:
    - print(f'{ip\_address} is not a valid IP address.')
      - continue
    - print(f'Octet {octet} is valid.')
- range() Function
  - The range() function in Python is used to generate a sequence of numbers within a given range. It's commonly used in for loops when there's a need to repeat an action a specific number of times.
  - The syntax of the range function is as follows:
    - range(start, stop, step)
    - start: (Optional) An integer number specifying at which position to start. Default is 0.
    - stop: An integer number specifying at which position to stop (not included).
    - step: (Optional) An integer number specifying the incrementation. Default is 1.
  - Example:
    - subnet = "192.168.1."
    - ip\_addresses = [subnet + str(i) for i in range(1, 256)]
    - 
    - for ip in ip\_addresses:
    - print(ip)
- else in for Loops
  - The else statement in a Python for loop specifies a block of code to be executed when the loop has finished, i.e., when all items in the sequence have been iterated over. **If the loop is exited prematurely with a break statement, the else block will not be executed.**
  - Here's the syntax:
    - for variable in iterable:
      - # code to be executed

- else:
  - # code to be executed after the loop has finished
- Example:
  - device\_list = ['Switch1', 'Router2', 'Firewall1']
  - search\_device = 'Router1'
  - 
  - for device in device\_list:
  - if device == search\_device:
  - print('Device found:', device)
  - break
  - else:
  - print('Device not found:', search\_device)
- pass Statement
  - The pass statement in Python is a placeholder statement that is used when the syntax requires a statement, but no action needs to be taken. It is often used in places where code will eventually go, but has not been written yet.
  - Here's the syntax:
    - for variable in iterable:
    - # code to be executed
    - pass # no action taken
  - Example
    - device\_list = ['Switch1', 'Router2', 'Firewall1']
    - 
    - for device in device\_list:
    - if device == 'Router2':
    - pass # no action taken for 'Router2'
    - else:
    - print('Device:', device)
- Example Problem
  - Company survey results include a department code, which has been stored in a list for counting responses by department.
  - 
  - Complete the function department\_count() to count the number of entries per department, returning a dictionary of department counts and return a count of invalid entries. The existing list department\_codes lists all valid codes. If an entry is not in department\_codes and is not the value "TEST", the entry is invalid. The returned dictionary should use the department code as the key and the count as the value.
  -



- Only the department count dictionary and invalid count returned by department\_count() will be graded for this assignment. The function should work for any list of codes passed to the function beyond the examples provided.
- Approach
  - def department\_count(entries):
  - department\_codes = ["HRD", "ENG", "MKT", "FIN", "IT"]
  - department\_counts = {'HRD': 0, 'ENG': 0, 'MKT': 0, 'FIN': 0, 'IT': 0}
  - invalid\_count = 0
  - 
  - #loops through entries
  - for entry in entries:
  - # checks if entry is in department codes
  - if entry in department\_codes:
  - department\_counts[entry] += 1
  - elif entry != "TEST":
  - invalid\_count += 1
  - 
  - # Remove zero-count departments (Safe approach)
  - keys\_to\_remove = [] # Temporary list to store keys to delete
  - for key in department\_counts:
  - if department\_counts[key] == 0:
  - keys\_to\_remove.append(key)
  - 
  - for key in keys\_to\_remove:
  - del department\_counts[key] # Delete keys after iteration
  - 
  - return department\_counts, invalid\_count
  - 
  - 
  - # You may alter the code below to view your return value(s).
  - # Only the generate\_users function will be graded for this assessment.
  - 
  - entries = ['HRD', 'MKT', 'HRD', 'IT', 'ENG', 'HRD', 'TEST', 'HRD', 'TEST', 'HRD', 'HRD', 'TEST', 'IT', 'TEST', 'HRD', 'TEST', 'IT', 'TEST', 'ENG', 'MKT', 'TEST', 'IT', 'IT', 'HRD', 'GUEST']
  - print(department\_count(entries))
  - 
  - # Expected return
  - # ({'HRD': 7, 'MKT': 2, 'IT': 5, 'ENG': 2}, 1)

## 11.2 Debugging and Fixing Errors

- Python programming bugs, also known as errors, are issues or problems that prevent a Python program from running correctly. They can occur for a variety of reasons and are typically classified into three categories:
  - Syntax Errors:
    - These occur when the Python parser is unable to understand a line of code. Syntax errors are usually the result of typos or misunderstandings about the Python language syntax. An example would be forgetting to close a parenthesis or misspelling a keyword.
  - Runtime Errors:
    - These errors occur during the execution of a program. They are often caused by operations that are mathematically illegal, such as division by zero, or by attempting to access a resource that isn't available, such as reading a file that doesn't exist.
  - Semantic Errors:
    - These are the most insidious errors as the program runs without crashing, but it doesn't produce the expected output. This could be due to an error in the program's logic.
- Python Debugging
  - Python debugging is a systematic process of finding and reducing the number of bugs or defects in a Python program, making it behave as expected. Here are some basic steps involved in Python debugging:
  - Understanding the Problem:
    - The first step in debugging is to understand the problem. This involves reproducing the error and analyzing the error message or incorrect output.
  - Isolating the Problem:
    - Once the problem is understood, the next step is to isolate the section of code causing the error. This can be done by commenting out sections of code or using print statements to check the values of variables at different stages of the program.
  - Using a Debugger:
    - Python comes with a built-in debugger called pdb. It allows stepping through the code line by line, inspecting variables, and setting breakpoints at specific lines of code.
  - Fixing the Error:
    - After identifying the cause of the error, the next step is to modify the code to fix the error. This could involve correcting a typo, changing a variable, or rewriting a section of code.
  - Testing the Solution:
    - After fixing the error, it's important to test the solution under different scenarios to ensure the error has been completely resolved.

- Common Debugging Techniques
  - Common debugging techniques in Python programming include:
    - Print Statements:
      - One of the simplest techniques is to use print statements to display the values of variables at certain points in the program. This can help identify unexpected values or behavior.
    - Using a Debugger:
      - Python's built-in debugger, pdb, allows stepping through the code line by line, inspecting variables, and setting breakpoints. This can be a powerful tool for understanding the flow of the program and identifying where things go wrong.
    - Code Review:
      - Sometimes, simply reviewing the code can help spot errors. This could be done individually or as part of a pair programming or code review session.
    - Unit Testing:
      - Writing unit tests can help catch errors and prevent regressions. Python's unittest module provides a framework for creating and running tests.
    - Logging:
      - For larger applications, using Python's logging module can provide valuable insights into the behavior of the program over time.
    - Profiling:
      - For performance issues, Python's cProfile module can help identify bottlenecks in the code.
- IDE Coding Errors
  - IDE coding errors are issues that arise when writing code in an Integrated Development Environment (IDE). These errors can be broadly classified into three categories:
    - Syntax Errors:
      - These are mistakes in the code's syntax, such as missing parentheses or incorrect indentation. IDEs often highlight these errors in real-time, allowing developers to correct them before running the program.
    - Runtime Errors:
      - These errors occur when the program is executed. Examples include dividing by zero or trying to access a non-existent file. Some IDEs provide debugging tools to help identify and resolve these errors.
    - Semantic Errors:

- These errors occur when the code compiles and runs without crashing, but it doesn't produce the expected results. This could be due to logic errors in the code. IDEs can't always catch these errors, so careful code review and testing are necessary.
  - In addition to these, IDEs can also flag linting errors. These are not necessarily errors, but rather suggestions for best practices or coding standards. Resolving these can make the code more readable and maintainable.
- Debugging Tools
  - Common debugging tools in Python programming include:
    - PDB:
      - The built-in Python debugger, pdb, allows developers to pause program execution, inspect variables, and step through the code.
    - PyCharm Debugger:
      - PyCharm, a popular Python IDE, comes with a powerful debugger that provides features like stepping through the code, breakpoints, and variable inspection.
    - Visual Studio Code Debugger:
      - Visual Studio Code (VS Code) is another popular IDE that includes a versatile debugger with support for remote debugging, multi-threaded debugging, and conditional breakpoints.
    - Logging:
      - Python's built-in logging module can be used to record the flow of the program and help identify issues.
    - Unit Testing Tools:
      - Tools like unittest, pytest, and doctest can help catch errors and prevent regressions.
    - Linters:
      - Tools like pylint and flake8 can catch potential issues in the code that might lead to errors.
    - Profiling Tools:
      - Tools like cProfile and memory\_profiler can help identify performance bottlenecks.

## 11.9 Input Validation

- Input validation is a crucial part of programming, especially in network automation where incorrect inputs can lead to significant issues. Here are some ways to validate function inputs in Python:
  - Type Checking
    - Ensure that the input is of the expected type.
    - `def connect_to_network_device(ip_address, username, password):`

- if not isinstance(ip\_address, str) or not isinstance(username, str) or not isinstance(password, str):
  - raise ValueError("All inputs must be strings")
- Value Checking
  - Check if the input values are within an expected range or format.
    - def validate\_ip(ip\_address):
    - parts = ip\_address.split('.')
      - if len(parts) != 4 or not all(part.isdigit() and 0 <= int(part) <= 255 for part in parts):
      - raise ValueError("Invalid IP address")
- Presence Checking
  - Ensure that necessary inputs are not missing.
    - def connect\_to\_network\_device(ip\_address=None, username=None, password=None):
    - if ip\_address is None or username is None or password is None:
    - raise ValueError("IP address, username, and password are required")
- Length Checking
  - Validate the length of the inputs.
    - def validate\_password(password):
    - if len(password) < 8:
    - raise ValueError("Password must be at least 8 characters long")
- Pattern Matching
  - Use regular expressions to match input patterns.
    - import re
    - def validate\_username(username):
    - if not re.match(r'^\w+\$', username):
    - raise ValueError("Username can only contain letters, numbers, and underscores")
- try/except Blocks
  - In Python, try/except blocks are used for exception handling. They allow the program to continue running even if an error or exception occurs.
    - try:
    - # Code that might raise an exception
    - except ExceptionType:
    - # Code to handle the exception
  - The try block contains the code that might raise an exception. Python will attempt to execute this code.

- If an exception is raised in the try block, the execution immediately moves to the except block.
- The ExceptionType is the type of exception that the except block can handle.
- If the exception type matches ExceptionType, then the code within the except block is executed.
- If no exception is raised in the try block, the except block is skipped.
- Here is an example:
  - try:
  - `x = 1 / 0` # This will raise a ZeroDivisionError
  - except ZeroDivisionError:
  - `x = 0` # This code will be executed when the exception is caught
- In this example, a ZeroDivisionError is raised when we try to divide by zero. The except block catches this exception and sets x to zero, allowing the program to continue running instead of crashing.
- Example:
  - `def connect_to_device(device):`
  - `# Simulate connecting to a device`
  - `if device == "bad_device":`
  - `raise Exception("Could not connect to device")`
  - `else:`
  - `return "Connected to device"`
  - 
  - `def send_command(connection, command):`
  - `# Simulate sending a command`
  - `if command == "bad_command":`
  - `raise Exception("Command failed")`
  - `else:`
  - `return "Command succeeded"`
  - 
  - `# Define device and command`
  - `device = "my_device"`
  - `command = "my_command"`
  - 
  - `try:`
  - `# Try to connect to the device and send a command`
  - `connection = connect_to_device(device)`
  - `result = send_command(connection, command)`
  - `print(result)`
  - `except Exception as e:`
  - `# If an exception occurs, print the error message`

- `print(f'An error occurred: {str(e)}')`

## 11.x Other Stuff Not Mentioned

- Types of Errors:
  - Runtime Error → Occurs during execution, e.g., trying to read a non-existent file.
  - Semantic Error → Code runs but produces unexpected output due to logic errors.
  - Syntax Error → Code structure is incorrect, e.g., missing colons in loops.
- Python Debugging Tools & Techniques:
  - pdb (Python Debugger) → Steps through code, inspects variables, sets breakpoints.
  - Profiling (cProfile) → Identifies performance bottlenecks in code.
  - Linters (pylint, flake8) → Detects potential issues and enforces coding standards.
- Common Python Debugging Scenarios:
  - Using `=` instead of `==` in conditionals
  - Modifying a list while iterating
  - Off-by-one errors
  - NameError
- Input Validation Techniques:
  - Pattern Matching (Regex) → Ensures a username contains only letters, numbers, and underscores

## 12.2 Working with Files in Python

- Example
  - Example:
  - `import paramiko`
  - 
  - `def automate_config(device_ip, username, password, config_file):`
  - `# Create an SSH client`
  - `ssh = paramiko.SSHClient()`
  - `ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())`
  - 
  - `# Connect to the network device`
  - `ssh.connect(device_ip, username=username, password=password)`
  - 
  - `# Open the configuration file`
  - `with open(config_file, 'r') as file:`
  - `commands = file.read().splitlines()`
  - 
  - `# Execute each command`
  - `for command in commands:`
  - `ssh.exec_command(command)`
  -

- # Close the connection
- ssh.close()

### 12.3 File Handling

- Commands
  - open(filename, mode): Opens a file in the specified mode ('r' for read, 'w' for write, 'a' for append, 'b' for binary). Returns a file object.
  - file.read([size]): Reads at most size bytes from the file. If size is not specified, it reads the whole file.
  - file.readline(): Reads the next line of the file.
  - file.readlines(): Returns a list of all lines in a file.
  - file.write(string): Writes the string to the file and returns the number of characters written.
  - file.close(): Closes the file.

### 12.4 Text vs Binary Files

- Text files and binary files are two types of files that can be handled in Python. Here are the key differences between them:
  - Text Files: These are human-readable files containing text (characters). They are structured as a sequence of lines, each containing a sequence of characters. This includes programming source code, HTML files, and more. When you open a text file in a text editor, it displays the contents as text.
  - Binary Files: These are not designed to be human-readable. They may contain any type of data encoded in binary form for computer processing, such as images, audio files, executable files, etc. When you open a binary file in a text editor, it often appears as a jumble of special characters.
- In Python, you can open both text and binary files using the open() function. The mode parameter determines how the file is opened: 't' for text (default) and 'b' for binary. For example, 'rb' opens a file for reading in binary mode, while 'wt' opens a file for writing text.

### 12.5 Text Files

- Working with text files in Python involves several steps:
  - Opening a File: Use the open() function with the filename and mode as arguments. The mode 'r' is for reading, 'w' for writing, and 'a' for appending.
  - Reading from a File: The read() method reads the entire file, readline() reads a single line, and readlines() reads all lines into a list.
  - Writing to a File: The write() method writes a string to the file.
  - Closing a File: The close() method closes the file, freeing up system resources.
- Example:
 

```
def read_config_commands(config_file):
    # Open the file in read mode
    file = open(config_file, 'r')
```



```

# Read all lines into a list
commands = file.readlines()

# Close the file
file.close()

# Remove newline characters
commands = [command.strip() for command in commands]

return commands

```

#### - Example 2

```

with open ("my_file.txt", "w") as f:
    f.write("ping\nlpconfig\nhostname\n")

```

```

def read_config_commands(config_file):
    # Open the file in read mode
    file = open(config_file, 'r')

    # Read all lines into a list
    commands = file.readlines()

    # Close the file
    file.close()

    # Remove newline characters
    commands = [command.strip() for command in commands]

    return commands

```

```

print(read_config_commands("my_file.txt"))

```

## 12.6 Binary Files

### - Commands

- Opening a File: Use the open() function with the filename and mode as arguments. The mode 'rb' is for reading in binary mode, and 'wb' is for writing in binary mode.
- Reading from a File: The read() method reads the entire file and returns the bytes.
- Writing to a File: The write() method writes bytes to the file.
- Closing a File: The close() method closes the file, freeing up system resources.

### - Example:

- def read\_firmware\_image(firmware\_file):
- # Open the file in binary read mode
- file = open(firmware\_file, 'rb')
- 
- # Read the entire file
- firmware\_data = file.read()
- 
- # Close the file
- file.close()
- 
- return firmware\_data
- Example 2:
- def read\_firmware\_image(firmware\_file):
- # Open the file in binary read mode
- file = open(firmware\_file, 'rb')
- 
- # Read the entire file
- firmware\_data = file.read()
- 
- # Close the file
- file.close()
- 
- return firmware\_data

## 12.7 Comma Separated Files (CSV)

- Comma Separated Values (CSV) files are a type of text file commonly used to store tabular data. Each line in the file represents a row in the table, and the values in each row are separated by commas.
- Python provides the csv module to read and write CSV files. Here's how to work with CSV files in Python:
  - Opening a File: Use the open() function with the filename and mode as arguments. The mode 'r' is for reading, and 'w' is for writing.
  - Creating a CSV Reader or Writer: Use the csv.reader() or csv.writer() function to create a reader or writer object.
  - Reading from a File: Use the next() function to read a row from the file. Each row is returned as a list of strings.
  - Writing to a File: Use the writerow() function to write a row to the file. The row should be a list of strings.
- Example:
  - import csv
  -

```

- def read_device_info(device_file):
-     # Open the file in read mode
-     file = open(device_file, 'r')
-
-     # Create a CSV reader
-     reader = csv.reader(file)
-
-     # Read the header row
-     headers = next(reader)
-
-     # Read the rest of the rows
-     devices = [row for row in reader]
-
-     # Close the file
-     file.close()
-
-     return headers, devices

```

## 12.8 Reading Files

- Reading Parts of a File
  - In Python, it's possible to read parts of a file using the `read()` method with a `size` argument, which specifies the number of bytes to read. If the `size` argument is not provided, the `read()` method reads the entire file
- Example:

```

def read_part_of_log(log_file, size):
    # Open the file in read mode
    file = open(log_file, 'r')

    # Read the specified number of bytes
    part_of_log = file.read(size)

    # Close the file
    file.close()

    return part_of_log

```

- Reading Lines in a File
  - In Python, reading a single line from a file can be accomplished using the built-in `open()` function and the `readline()` method.
    - with `open('filename.txt', 'r')` as `file`:
    - `first_line = file.readline()`

## 12.9 Writing to Files

- In Python, writing to an existing file can be done using the built-in `open()` function with the 'a' or 'w' mode. The 'a' mode appends to the end of the file, while the 'w' mode overwrites the file. Here's an example:
  - with `open('filename.txt', 'a')` as file:
    - `file.write('New line to append\n')`
  - In this code, 'filename.txt' should be replaced with the path to the file you want to write to. The string 'New line to append\n' is the content to be written to the file.
- Example:
  - Consider a case where there's a need to log network device status to a file:
 

```
import os

with open('device_ip.txt', 'r') as file:
    device_ip = file.readline().strip()

response = os.system("ping -c 1 " + device_ip)

with open('log.txt', 'a') as file:
    if response == 0:
        file.write(device_ip + ' is up!\n')
    else:
        file.write(device_ip + ' is down!\n')
```
- Creating a New File
  - In Python, creating a new file can be done using the built-in `open()` function with the 'x', 'a', or 'w' mode. Here's how each mode works:
    - 'x': Creates a new file and opens it for writing. If the file already exists, the operation fails.
    - 'a': Opens the file for writing, appending to the end of the file if it exists.
    - 'w': Opens the file for writing. If the file exists, it is truncated. If the file does not exist, it is created.
  - Here's an example of creating a new file with each mode:
    - # 'x' mode
    - try:
      - with `open('newfile_x.txt', 'x')` as file:
      - `file.write('Content for the new file\n')`
    - except `FileExistsError`:
    - `print('File already exists.')`
    - 
    - # 'a' mode
    - with `open('newfile_a.txt', 'a')` as file:

- file.write('Content for the new file\n')
- 
- # 'w' mode
- with open('newfile\_w.txt', 'w') as file:
- file.write('Content for the new file\n')
- Consider a case where there's a need to log network device status to a new file:
  - import os
  - 
  - with open('device\_ip.txt', 'r') as file:
  - device\_ip = file.readline().strip()
  - 
  - response = os.system("ping -c 1 " + device\_ip)
  - 
  - try:
  - with open('new\_log.txt', 'x') as file:
  - if response == 0:
  - file.write(device\_ip + ' is up!\n')
  - else:
  - file.write(device\_ip + ' is down!\n')
  - except FileExistsError:
  - print('Log file already exists.')

## 12.10 Closing a File

- In Python, closing a file can be done using the close() method. Here's an example:
 

```
file = open('filename.txt', 'r')
# Perform file operations
file.close()
```
- Closing files in Python is necessary for several reasons:
  - It frees up system resources that were tied up with the file.
  - It ensures that changes made to the file are saved. Some changes made to a file in Python may not be immediately written to disk; closing the file ensures that these changes are not lost.
  - It prevents further modifications to the file. Once a file is closed, attempting to write to it will result in an error.
- Consider a case where there's a need to log network device status to a file and then close it:
  - import os
  - 
  - file = open('device\_ip.txt', 'r')
  - device\_ip = file.readline().strip()
  - file.close()

- 
- response = os.system("ping -c 1 " + device\_ip)
- 
- file = open('log.txt', 'a')
- if response == 0:
- file.write(device\_ip + ' is up!\n')
- else:
- file.write(device\_ip + ' is down!\n')
- file.close()
- With Statement is more efficient
  - import os
  - 
  - with open('device\_ip.txt', 'r') as file:
  - device\_ip = file.readline().strip()
  - 
  - response = os.system("ping -c 1 " + device\_ip)
  - 
  - with open('log.txt', 'a') as file:
  - if response == 0:
  - file.write(device\_ip + ' is up!\n')
  - else:
  - file.write(device\_ip + ' is down!\n')

#### 12.11 Checking for File Existence

- In Python, checking if a file exists can be done using the os.path.exists() function from the os module. Here's an example:
  - import os
  - 
  - # Specify the file path
  - file\_path = 'filename.txt'
  - 
  - # Check if the file exists
  - if os.path.exists(file\_path):
  - print('The file exists.')
  - else:
  - print('The file does not exist.')
- Example:
  - Consider a case where there's a need to check if a configuration file for a network device exists:
    - import os
    -

- # Specify the file path
- config\_file\_path = 'config.txt'
- 
- # Check if the configuration file exists
- if os.path.exists(config\_file\_path):
- print('The configuration file exists.')
- else:
- print('The configuration file does not exist.')
- 12.12 Deleting Files and Folders
  - Deleting Files
    - In Python, deleting a file can be done using the remove() function from the os module. Here's an example:
      - import os
      - 
      - # Specify the file path
      - file\_path = 'filename.txt'
      - 
      - # Check if the file exists
      - if os.path.exists(file\_path):
      - # Delete the file
      - os.remove(file\_path)
      - else:
      - print('The file does not exist.')
  - Example:
    - Consider a case where there's a need to delete a log file after it has been processed:
      - import os
      - 
      - # Specify the file path
      - log\_file\_path = 'log.txt'
      - 
      - # Check if the log file exists
      - if os.path.exists(log\_file\_path):
      - # Delete the log file
      - os.remove(log\_file\_path)
      - else:
      - print('The log file does not exist.')
  - Deleting a Folder
    - In Python, deleting a folder can be done using the rmdir() function from the os module. Here's an example:

- import os
- 
- # Specify the folder path
- folder\_path = 'foldername'
- 
- # Check if the folder exists
- if os.path.exists(folder\_path):
- # Delete the folder
- os.rmdir(folder\_path)
- else:
- print('The folder does not exist.')
- Example:
- Consider a case where there's a need to delete a folder containing logs for a network device after they have been processed:
  - import os
  - 
  - # Specify the folder path
  - log\_folder\_path = 'logs'
  - 
  - # Check if the log folder exists
  - if os.path.exists(log\_folder\_path):
  - # Delete the log folder
  - os.rmdir(log\_folder\_path)
  - else:
  - print('The log folder does not exist.')

### 12.13 Comparing Files

- The filecmp module in Python is a utility for comparing files and directories. It provides functions to compare files and directories, and to report detailed information about the differences. Its primary use is in searching for duplicate files and in comparing directory trees.
- Here is an example of network automation using the filecmp module:
 

```
# Compare two configuration files
if filecmp.cmp('/path/to/config1.txt', '/path/to/config2.txt'):
    print('The configuration files are the same.')
else:
    print('The configuration files are different.')
```
- In this example, filecmp.cmp('/path/to/config1.txt', '/path/to/config2.txt') compares two configuration files. If the files are the same, it prints 'The configuration files are the same.' If the files are different, it prints 'The configuration files are different.' This can be useful in network automation tasks, such as checking if a network device's



configuration has changed. Note that the filecmp module automatically handles the intricacies of file comparison, allowing network engineers to focus on the automation task at hand, rather than the details of file comparison.

## 13.2 Python Scripts, Modules, Packages, and Libraries

- Python Scripts
  - A Python script is a file containing code written in the Python programming language. It automates tasks that would otherwise be executed line by line in Python's interactive mode. The script, saved with a .py extension, can be run from the command line. It's widely used in various fields, including network automation, due to Python's simplicity and the availability of numerous libraries. Python scripts can interact with network devices, execute commands, and retrieve data, making network management more efficient.
- Python Modules
  - A Python module is a file containing Python definitions and statements. It provides a way to logically organize Python code. Functions, classes, or variables defined in a module can be imported into other modules or scripts using the import statement.
  - For network automation, some commonly used Python modules are:
    - Netmiko: For SSH connections to routers and switches
    - NAPALM: Provides a set of functions to interact with different network device Operating Systems
    - Ansible: An open-source software provisioning, configuration management, and application-deployment tool
    - Paramiko: For implementing SSHv2 protocol
    - Exscript: Automates Telnet and SSH connections to remote hosts.
- Python Packages
  - A Python package is a collection of Python modules, or files containing Python code, organized in a directory hierarchy. It allows for a structured, modular approach to programming. Packages are installed using package managers like pip.
- Python Libraries
  - A Python library is a reusable chunk of code that you may want to include in your programs/projects. It provides pre-written functionality, reducing the amount of code you need to write. In the context of network automation, Python libraries offer pre-built modules for common networking tasks. Some of these libraries include:
    - Requests: For making HTTP requests.
    - BeautifulSoup: For parsing HTML and XML documents.
    - Scapy: For crafting, sending and receiving packets.

## 13.3 Python Modules

- Create a Module
  - Creating a Python module involves writing a Python script with functions, classes, or variables, and saving it with a .py extension. This file can then be imported into other Python scripts using the import statement. For example, if a file named mymodule.py contains a function my\_function(), it can be imported and used in another script as follows:
    - import mymodule
    - mymodule.my\_function()
  - This allows for code reuse across multiple scripts, improving code organization and readability.
- Using a Module
  - Creating a Python module involves writing a Python script with functions, classes, or variables, and saving it with a .py extension. This file can then be imported into other Python scripts using the import statement.
  - Consider a module named network.py with a function ping\_device(ip\_address). This function could use the os library to ping a device:
    - import os
    - 
    - def ping\_device(ip\_address):
    - response = os.system("ping -c 1 " + ip\_address)
    - if response == 0:
    - return True
    - else:
    - return False
  - This module can be used in another script as follows:
    - import network
    - 
    - ip = '192.168.1.1'
    - if network.ping\_device(ip):
    - print(f'Device {ip} is online.')
    - else:
    - print(f'Device {ip} is offline.')
  - This script imports the network module and uses its ping\_device function to check if a device is online. This demonstrates how modules can be used to organize and reuse code in network automation tasks.
- Renaming a Module
  - In Python, a module can be renamed when it's imported using the as keyword. This allows for more convenient or intuitive naming.
  - Consider a network automation module named network.py with a function ping\_device(ip\_address). It can be renamed upon import as follows:

- import network as net
- 
- ip = '192.168.1.1'
- if net.ping\_device(ip):
- print(f'Device {ip} is online.')
- else:
- print(f'Device {ip} is offline.')
- In this script, network is renamed to net, making subsequent calls to the module's functions more concise. This is particularly useful when dealing with modules that have long or complex names.
- Built-In Modules
  - Built-in modules in Python are libraries that come pre-installed with Python. They provide functions and classes for a variety of tasks without the need for additional installation.
  - For example, the os and socket modules are often used in network automation. The os module provides functions for interacting with the operating system, while the socket module is used for network communications.
  - Here's an example of using these modules to get the hostname and IP address:
    - import socket
    - import os
    - 
    - hostname = socket.gethostname()
    - ip\_address = socket.gethostbyname(hostname)
    - 
    - print(f'Hostname: {hostname}')
    - print(f'IP Address: {ip\_address}')
  - In this script, socket.gethostname() gets the host name and socket.gethostbyname(hostname) gets the IP address of the host. The os module could be used for tasks like changing directories or running system commands.
- List Function Names
  - To list all function names in a Python module, the dir() function can be used. It returns a list of names in the current local scope or a list of attributes of an object. When a module is passed as an argument to dir(), it returns a list of the module's attributes, including its functions. Here's an example:
    - import math
    - functions = [name for name in dir(math) if callable(getattr(math, name))]
    - print(functions)
- Import from a Module

- In Python, specific parts of a module can be imported using the `from ... import ...` statement. This allows for importing only the necessary functions or classes, making the code more efficient.
- Consider a network automation task that requires the ping function from a module named `network_tools.py`. Instead of importing the entire module, only the ping function can be imported as follows:
  - `from network_tools import ping`
  - 
  - `ip = '192.168.1.1'`
  - `if ping(ip):`
  - `print(f'Device {ip} is online.')`
  - `else:`
  - `print(f'Device {ip} is offline.')`
- The `ipaddress` Module
  - The `ipaddress` module in Python is a powerful tool for manipulating and analyzing IP addresses and networks. It provides classes for handling IPv4 and IPv6 addresses, networks, and interfaces. These classes support validation, comparison, sorting, and other operations.
  - Example:
    - `import ipaddress`
    - 
    - `# Define the network`
    - `network = ipaddress.ip_network('192.0.2.0/24')`
    - 
    - `# Iterate over all hosts in the network`
    - `for host in network.hosts():`
    - `print(host)`
- The `help` Module
  - The `help()` function in Python is a built-in function that can be used to access the built-in documentation for Python modules, functions, classes, keywords, etc. This function is most commonly used in the Python interpreter, and it is a very useful tool for understanding and using different Python functionalities.
  - Here is an example of how to use the `help()` function:
    - `help(print)`
  - This will display the documentation for the `print` function. The output will include a brief description of the function, its syntax, the arguments it takes, and what it returns.
  - The `help()` function can also be used on Python modules. For example:
    - `import os`
    - `help(os)`

## 13.4 Python Libraries

- Install Library
  - Python libraries can be installed using package managers like pip. The command `pip install library_name` is used in the command line interface. For instance, to install the netmiko library, which is commonly used in network automation, the command would be `pip install netmiko`. It's recommended to use a virtual environment to avoid conflicts between libraries. If the library is already installed, it can be upgraded using `pip install --upgrade library_name`. Always ensure pip is upgraded to the latest version before installing libraries.
- Standard Library
  - The Python Standard Library is a collection of modules and packages that come pre-installed with Python. It provides a wide range of functionalities, including mathematical operations, file I/O, system calls, and even Internet protocols. This library is considered "standard" because it's available in every Python installation. It's designed to enhance Python's usability and standardize solutions for common programming tasks, reducing the need for third-party libraries. The Python Standard Library is a key reason for Python's popularity, as it simplifies many complex tasks in areas like network automation, data analysis, and web development.
- Running a Library
  - Running a Python library involves importing it into a Python script and then calling its functions or classes. For instance, the socket library, which is part of the Python Standard Library, can be used for network automation tasks.
  - Here's an example of using it to establish a TCP connection:
    - `import socket`
    - 
    - `# Create a socket object`
    - `s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
    - 
    - `# Define the server and the port`
    - `server = 'hostname'`
    - `port = 80`
    - 
    - `# Connect to the server`
    - `s.connect((server, port))`

## 13.5 Python Packages

- Listing a Package
  - Listing installed Python packages can be done using pip, the Python package installer. The command `pip list` is used in the command line interface, which returns a list of all installed packages along with their versions. To check if a

specific package is installed, `pip show package_name` can be used. If the package is installed, this command will display information about it. If not, it will not return any output. These commands help manage and track the packages used in a Python environment.

- Installing a Package

- Python packages can be installed using pip, the Python package installer. The command `pip install package_name` is used in the command line interface. For instance, to install the netmiko package, which is commonly used in network automation, the command would be `pip install netmiko`. It's recommended to use a virtual environment to avoid conflicts between packages. If the package is already installed, it can be upgraded using `pip install --upgrade package_name`. Always ensure pip is upgraded to the latest version before installing packages.

- Using a Package

- To use a Python package, it must first be imported into the Python script using the import statement. Once imported, the functions, classes, or variables defined in the package can be accessed using the dot notation.
- Example:
  - Consider the netmiko package, which is commonly used in network automation. Here's an example of using it to establish an SSH connection to a Cisco device:

```
from netmiko import ConnectHandler

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

connection = ConnectHandler(**device)

output = connection.send_command('show ip int brief')
print(output)

connection.disconnect()
```

- Removing a Package

- Python packages can be removed using pip, the Python package installer. The command `pip uninstall package_name` is used in the command line interface. For instance, to uninstall the netmiko package, which is commonly used in network

automation, the command would be `pip uninstall netmiko`. This command will ask for confirmation before proceeding with the uninstallation. To skip the confirmation, use the `-y` option: `pip uninstall -y package_name`. Always ensure `pip` is upgraded to the latest version before uninstalling packages. Note that uninstalling a package will remove it from the system, and it will no longer be available for import in Python scripts.

#### 14.1 Utilizing APIs

- Application Programming Interfaces (APIs) are sets of rules and protocols for building software and applications. They allow different software systems to communicate and share data. Interfacing with APIs using Python is beneficial as it enables automation, data extraction, and interaction with online services and resources, enhancing the functionality and efficiency of Python applications.
- Let's consider a network automation task of updating the configuration of a network device using an API. Here's how Python can be used for this task:
  - `import requests`
  - `import json`
  - 
  - `# Step 1: Read data from the API`
  - `url = "http://network-device-api/config"`
  - `response = requests.get(url)`
  - `data = response.json()`
  - 
  - `# Step 2: Store the data as a file`
  - `with open('config.json', 'w') as file:`
  - `json.dump(data, file)`
  - 
  - `# Step 3: Manipulate the data`
  - `data['setting'] = 'new value'`
  - 
  - `# Step 4: Store the updated data as a file`
  - `with open('config.json', 'w') as file:`
  - `json.dump(data, file)`
  - 
  - `# Step 5: Push the changes back to the API`
  - `response = requests.put(url, data=json.dumps(data))`
- Example
  - `import requests`
  - 
  - `def get_posts():`
  - `response = requests.get('https://jsonplaceholder.typicode.com/posts')`

- if response.status\_code == 200:
- posts = response.json()
- for post in posts:
- print(f'Post ID: {post['id']}, Title: {post['title']}")
- 
- get\_posts()
- 14.2 Network Log Analysis Example
  - Scenario:
    - Given a csv file, analyze log data from various sources such as firewalls, intrusion detection systems, and other security appliances. Filter the logs to identify patterns and suspicious activities, then group the logs by user to count their activities.
  - Here's a Python code snippet that uses the pandas library to analyze log data from a CSV file:
    - import pandas as pd
    - 
    - # Load the CSV file
    - df = pd.read\_csv('logs.csv')
    - 
    - # Filter logs to identify patterns and suspicious activities
    - suspicious\_df = df[df['activity'] == 'suspicious']
    - 
    - # Group logs by user and count their activities
    - user\_activity\_count = suspicious\_df.groupby('user').size()
  - This code first loads the log data from a CSV file into a DataFrame. It then filters the logs to identify suspicious activities. Finally, it groups the logs by user and counts their activities. Please replace 'logs.csv', 'activity', 'suspicious', and 'user' with the actual CSV file name, activity column name, suspicious activity identifier, and user column name, respectively. Also, error checking is omitted for brevity. In a real-world scenario, each operation should be checked for errors.
  - 
  - Note: a DataFrame is a two-dimensional data structure that organizes data into rows and columns, much like a spreadsheet. It's available in languages like Python and R, and is commonly used in data analysis. In Python, the pandas library provides the DataFrame, which can handle heterogeneous data and allows for flexible manipulation of data

### 16.1 Practice Lab 1

- An outdated design file has stored red, blue, and green (RGB) color values as separate integer values within a list. Combine all three color values into a single string value properly formatted as an RGB color.



- 
- Complete the Python function `format_rgb`. The function should accept a list of integers representing separate RGB values, combine the RGB values into a single string value properly formatted as an RGB color, and return the formatted RGB color. A formatted RGB color does not contain space characters between values.
- 
- Only the `format_rgb` function will be graded for this assessment. The function should work for any list of integers passed to `format_rgb` beyond the examples provided.
- 
- Example: If the stored RGB values are
  - `[255, 165, 13]`
  - the expected return is
  - 
  - `rgb(255,165,13)`
- Example: If the stored RGB values are
  - `[0, 0, 0]`
  - the expected return is
  - 
  - `rgb(0,0,0)`
- My Approach
  - `def format_rgb(rgb):`
  - `return f"rgb({rgb[0]},{rgb[1]},{rgb[2]})"`

## 16.2 Practice Lab 2

- Complete the Python function `minutes_to_hours`. The function should accept an integer representing the execution time of a process in minutes, convert the value from minutes to hours, and return a float representing the execution time in hours. There are 60 minutes in an hour. The function should utilize float division and not integer division.
- 
- Example: If the argument representing 60 minutes is
  - 60 the expected return is 1.0
  - Example: If the argument representing 30 minutes is
  - 30 the expected return is 0.5
- My Approach
  - `def minutes_to_hours(minutes):`
  - `return minutes / 60.0`

## 16.3 Practice Lab 3

- A list of application logs have been collected. Each log's details are stored in a dictionary with the keys `app`, `level`, `message`, and `timestamp`. Updates to the log details are needed based on the following requirements:

- 
- Change the log level to "ERROR" for the log with application name webserver.
- Update the timestamp to "2023-12-07T12:30:00" for the log with application name database.
- Complete the Python function `update_log_list`. The function should accept a list of dictionaries representing log files, update the values in the log files based on the two requirements, and return the updated list of log files.
  
- Example: If the stored log files are
  - `log_sample = [`
  - `{"app": "webserver", "level": "INFO", "message": "System error", "timestamp": "2023-12-07T12:25:00"},`
  - `{"app": "database", "level": "WARN", "message": "High CPU usage", "timestamp": "2023-12-07T12:20:00"}]`
  - the expected return is
  - 
  - `[{'app': 'webserver', 'level': 'ERROR', 'message': 'System error', 'timestamp': '2023-12-07T12:25:00'}, {'app': 'database', 'level': 'WARN', 'message': 'High CPU usage', 'timestamp': '2023-12-07T12:30:00'}]`
- Example: If the stored log files are
  - `log_sample = [`
  - `{"app": "webserver", "level": "ERROR", "message": "Critical error", "timestamp": "2023-12-07T11:55:00"},`
  - `{"app": "database", "level": "ERROR", "message": "Database connection lost", "timestamp": "2023-12-07T11:50:00"}]`
  - the expected return is
  - 
  - `[{'app': 'webserver', 'level': 'ERROR', 'message': 'Critical error', 'timestamp': '2023-12-07T11:55:00'}, {'app': 'database', 'level': 'ERROR', 'message': 'Database connection lost', 'timestamp': '2023-12-07T12:30:00'}]`
- My Approach
  - `def update_log_list(log_list):`
  - `for dic in log_list:`
  - `if dic["app"] == "webserver":`
  - `dic["level"] = "ERROR"`
  - `if dic["app"] == "database":`
  - `dic["timestamp"] = "2023-12-07T12:30:00"`
  - `return log_list`

## 16.4 Practice Lab 4

- A business requires that each employee ID begins with a department identifier and ends with an individual identifier (e.g., "HRD00123", "ENG00567"). The department identifier is a **3-letter department code in all uppercase**. The individual identifier is a **5-digit numeric value**.
- Complete the Python script to create a custom function name `validate_id`. The function should accept a string parameter representing an employee ID, determine if the ID meets the requirements, and return a Boolean value, with `True` returned if all requirements are met and `False` returned if any requirement is not met.
- Example: If the string representing an employee ID is
  - HRD00123 the expected return is `True`
- Example: If the string representing an employee ID is
  - Ops123456 the expected return is `False`
- My Approach
  - ```
def validate_id(id):
    if len(id) == 8:
        if id[:3].upper() == id[:3] and id[3:8].isdigit():
            return True
        else:
            return False
    else:
        return False
```
- Most Optimal Approach
  - ```
def validate_id(emp_id):
    return len(emp_id) == 8 and emp_id[:3].isupper() and emp_id[3:].isdigit()
```

## 16.5 Practice Lab 5

- A Python function `same_subnet` converts IP addresses and subnet mask to binary strings in order to determine if two IPv4 addresses are in the same subnet, returning a string indicating whether both IP addresses are in the same subnet or not.
- Complete the Python function `same_subnet`. The function should accept three strings representing two IP addresses and a subnet mask, determine if both IP addresses are in the same subnet, and return a predefined string statement indicating if both IP addresses are in the same subnet.
- Only the `same_subnet` function will be graded for this assessment. The function should work for any IP addresses or subnet masks passed to `same_subnet` beyond the examples provided.

- Example: If the two IP addresses and subnet mask are
  - '192.168.1.100', '192.168.1.200', '255.255.255.0'
  - the expected return is 192.168.1.100 and 192.168.1.200 are in the same subnet
- Example: If the two IP addresses and subnet mask are
  - '192.168.1.100', '192.168.2.200', '255.255.255.0'
  - the expected return is 192.168.1.100 and 192.168.2.200 are not in the same subnet
- My Approach
  - def same\_subnet(ip1, ip2, subnet\_mask):
  - # convert IP addresses to binary strings
  - ip1\_bin = ".join([format(int(x), '08b') for x in ip1.split('.')])
  - ip2\_bin = ".join([format(int(x), '08b') for x in ip2.split('.')])
  - 
  - # convert subnet mask to binary string
  - subnet\_bin = ".join([format(int(x), '08b') for x in subnet\_mask.split('.')])
  - 
  - # get network address portion for both IP addresses
  - subnet\_len = subnet\_bin.count('1')
  - network1\_bin = ip1\_bin[:subnet\_len]
  - network2\_bin = ip2\_bin[:subnet\_len]
  - 
  - # Predefined statement indicating if IP addresses are in the same subnet
  - x = f" {ip1} and {ip2} are in the same subnet"
  - y = f" {ip1} and {ip2} are not in the same subnet"
  - 
  - return x if network1\_bin == network2\_bin else y

## 16.6 Practice Lab 6

- A list of float values measuring the percentage of CPU usage for servers has been collected. Servers with CPU usage greater than 90% should be flagged.
- Complete the Python function identify\_high\_cpu. The function should accept a list of floats representing the percentage of CPU usage for servers, determine which servers have higher than 90% CPU usage, and return the list of high-usage servers by index value.
- Example: If the stored CPU percentages are
  - [85.0, 92.5, 88.0, 95.2] the expected return is [1, 3]
- Example: If the stored CPU percentages are

- [91.0, 88.8, 89.5] the expected return is [0]
- My Approach
  - `def identify_high_cpu(cpu_list):`
  - `overuse = list()`
  - `i = 0`
  - `while i < len(cpu_list):`
  - `if cpu_list[i] > 90.0:`
  - `overuse.append(i)`
  - `i += 1`
  - `else:`
  - `i += 1`
  - `return overuse`
- Most Optimal
  - `def identify_high_cpu(cpu_list):`
  - `return [i for i, usage in enumerate(cpu_list) if usage > 90.0]`

#### 16.7 Practice Lab 7

- An existing function `line_count` is meant to open a text file, read the contents, and return the number of lines in the file. Several existing issues throw errors, and the function is not working as intended.
- Update the code within the Python function `line_count`. The function should accept a string identifying the name of a text file, read the contents of the text file, determine the number of lines in the text file, and return the number of lines in the text file. For simplicity, assume each line except the last ends with a newline character.
- Only the `line_count` function will be graded for this assessment. The function should work for any text file passed to `line_count` beyond the examples provided.
- Example: If the text file "test.txt" contains
  - Line 1
  - Line 2
  - Line 3
  - the expected return is 3
- Example: If the text file "lorem.txt" contains
  - Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  - Pellentesque tincidunt velit non iaculis porttitor.
  - Phasellus mattis, metus non posuere mollis, eros augue dictum.
  - Quisque porttitor est nec eros maximus, id sodales.

- the expected return is 4

## 16.8 Practice Lab 8

- Complete the python function named `write_dict_to_csv` that appends predefined data to an existing file `config.csv` using the CSV library.
- Note:
  - There should be a newline after the last row of data per CSV library defaults.
  - Only the file output will be graded for this assessment, standard output will be ignored.
  - Print to check your work; what you print to stdout does not affect the assessment
- Here is an example of the usage:
 

```
write_dict_to_csv('config.csv')
```
- Here is an example of what the `config.csv` should look like when written:
 

```
device_name,ip_address
Router1,192.168.1.1
Router2,192.168.1.2
```
- Optimal Approach
 

```
import csv
```

```
def write_dict_to_csv(filename):
    # Data to be written to the CSV file
    fieldnames = ['device_name', 'ip_address']
    data = [
        {'device_name': 'Router1', 'ip_address': '192.168.1.1'},
        {'device_name': 'Router2', 'ip_address': '192.168.1.2'}
    ]

    with open(filename, 'a') as file:
        writer = csv.DictWriter(file, fieldnames=fieldnames)

        writer.writeheader()
        writer.writerows(data)
```

## 16.9 Practice Lab 9

- A database contains user data uploads with varying submission times. To process the latest information first, a function is needed to identify the most recent upload time.
- Complete the Python function `find_latest`. The function should accept an unordered list of user upload time strings, convert each string value into a datetime object using the provided `date_format` pattern, and return the most recent upload time as a datetime object.

- Example: If the stored user upload values are
  - ['12/15/2023 08:45 AM', '12/14/2023 03:30 PM', '12/16/2023 11:20 AM', '12/13/2023 06:15 PM']
  - the expected return is 2023-12-16 11:20:00
- Example: If the stored user upload values are
  - ['12/20/2023 02:00 AM', '12/18/2023 10:30 AM', '12/16/2023 07:45 PM', '12/14/2023 04:15 PM']
  - the expected return is 2023-12-20 02:00:00
- My Approach
  - from datetime import datetime
  - 
  - def find\_latest(submissions):
  - # Specify a date format
  - date\_format = '%m/%d/%Y %I:%M %p'
  - 
  - datetime\_list = list()
  - # Convert string values into datetime objects.
  - for time in submissions:
  - datetime\_list.append(datetime.strptime(time, date\_format))
  - 
  - # Determine and return latest
  - return max(datetime\_list)
- Optimal Approach
  - def find\_latest(submissions):
  - # Specify the date format (matches the given string format)
  - date\_format = '%m/%d/%Y %I:%M %p'
  - 
  - # Convert string values into datetime objects
  - datetime\_objects = [datetime.strptime(timestamp, date\_format) for timestamp in submissions]
  - 
  - # Determine and return the latest datetime
  - return max(datetime\_objects)

## 16.10 Practice Lab 10

- A bug has been submitted for a Python function named scan\_ports. The function is supposed to use the socket library to scan the loopback address and check if ports 0-x are open/closed and return a list of tuples with the port status. Correct the logical errors in the code.
- Here is an example of the usage:


```
print(scan_ports(5))
```

- Here is an example of the expected output:

```
[(0, 'OPEN'), (1, 'OPEN'), (2, 'OPEN'), (3, 'OPEN'), (4, 'OPEN'), (5, 'OPEN')]
```

- import socket
- import time
- import unittest.mock as mock
- 
- def scan\_ports(x):
- # Mock socket
- with mock.patch('socket.socket') as mock\_socket:
- mock\_socket.return\_value.connect\_ex.return\_value = 0 # do not edit
- target\_IP = '127.0.0.1'
- 
- # Instantiate a socket object.
- s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)
- 
- # Set a timeout for the socket operations.
- s.settimeout(5)
- 
- # Create an empty list to store port status as a list tuples.
- open\_ports = []
- 
- # Iterate over ports checking the connection and appending the port status to the list.
- for i in range(0, x+1):
- conn = s.connect\_ex((target\_IP, i))
- if(conn == 0):
- open\_ports.extend(open\_ports + [i, 'CLOSED'])
- else:
- open\_ports.extend(open\_ports + [i, 'OPEN'])
- 
- # Close the socket connection.
- s.close()
- 
- # Return the list of open ports as a list of tuples.
- return open\_ports
- 
- # You may alter the code below to test your solution or print help documentation.
- # Only the scan\_ports function will be graded for this assessment.
- 
- # print(scan\_ports(5))



- # help(socket.socket)
- # help(socket.socket.connect\_ex)
  
- My Approach
  - import socket
  - import unittest.mock as mock
  - 
  - def scan\_ports(x):
  - # Mock socket
  - with mock.patch('socket.socket') as mock\_socket:
  - mock\_socket.return\_value.connect\_ex.return\_value = 0 # Do not edit
  - target\_IP = '127.0.0.1'
  - 
  - # Create an empty list to store port status as a list of tuples
  - port\_status = []
  - 
  - # Iterate over ports checking the connection and appending the port status
  - for i in range(0, x + 1):
  - s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # Create a
  - new socket for each connection
  - s.settimeout(5) # Set timeout
  - 
  - conn = s.connect\_ex((target\_IP, i)) # Check port connection
  - 
  - # Append correct port status as a tuple
  - port\_status.append((i, "OPEN" if conn == 0 else "CLOSED"))
  - 
  - s.close() # Close socket after checking the port
  - 
  - # Return the list of open ports as a list of tuples
  - return port\_status
  - 
  - #  Test the function
  - print(scan\_ports(5))
  - # Expected Output: [(0, 'OPEN'), (1, 'OPEN'), (2, 'OPEN'), (3, 'OPEN'), (4,
  - 'OPEN'), (5, 'OPEN')]
  -