**Workshop Outline**

**Objective :** The workshop is designed to provide Basic knowledge of Python and Junos PyEz

**Part I : Intro to Automation**

**- Network Automation**

**Part II : Basic Python**

**- What is & How to install and setup Python?**

**- Python Modules for Network Engineer**

**- Python Basic**

**- Keyword , Identifiers and Indentation**

**- Comments & Multi-line comments**

**- Variables**

**- Input, Output and Import**

**- Operators**

**- Flow Control (If & Loop)**

**- Exception**

**- Function**

**- Class**

**- Python practices**

**Part III : Junos PyEZ Overview**

**- What is it?**

**- Setting Up Junos PyEZ Managed Nodes**

**- Connect to devices, retrieve facts with JUNOS PyEZ**

**- Using Junos PyEZ to Retrieve Facts**

**- Config Utility to Configure Devices Running Junos OS**

**- Methods defined in the class config**

**- Loading Configuration Data from a Local or Remote File**

**- Rolling Back the Configuration**

**- Loading the Rescue Configuration**

**- Committing the Configuration**

**- Lab Practices**

**Automation Technology Overview**

Automation is the technology by which a process or procedure is performed with minimal human assistance. It is the use of various control systems for operating equipment

**Network Automation**

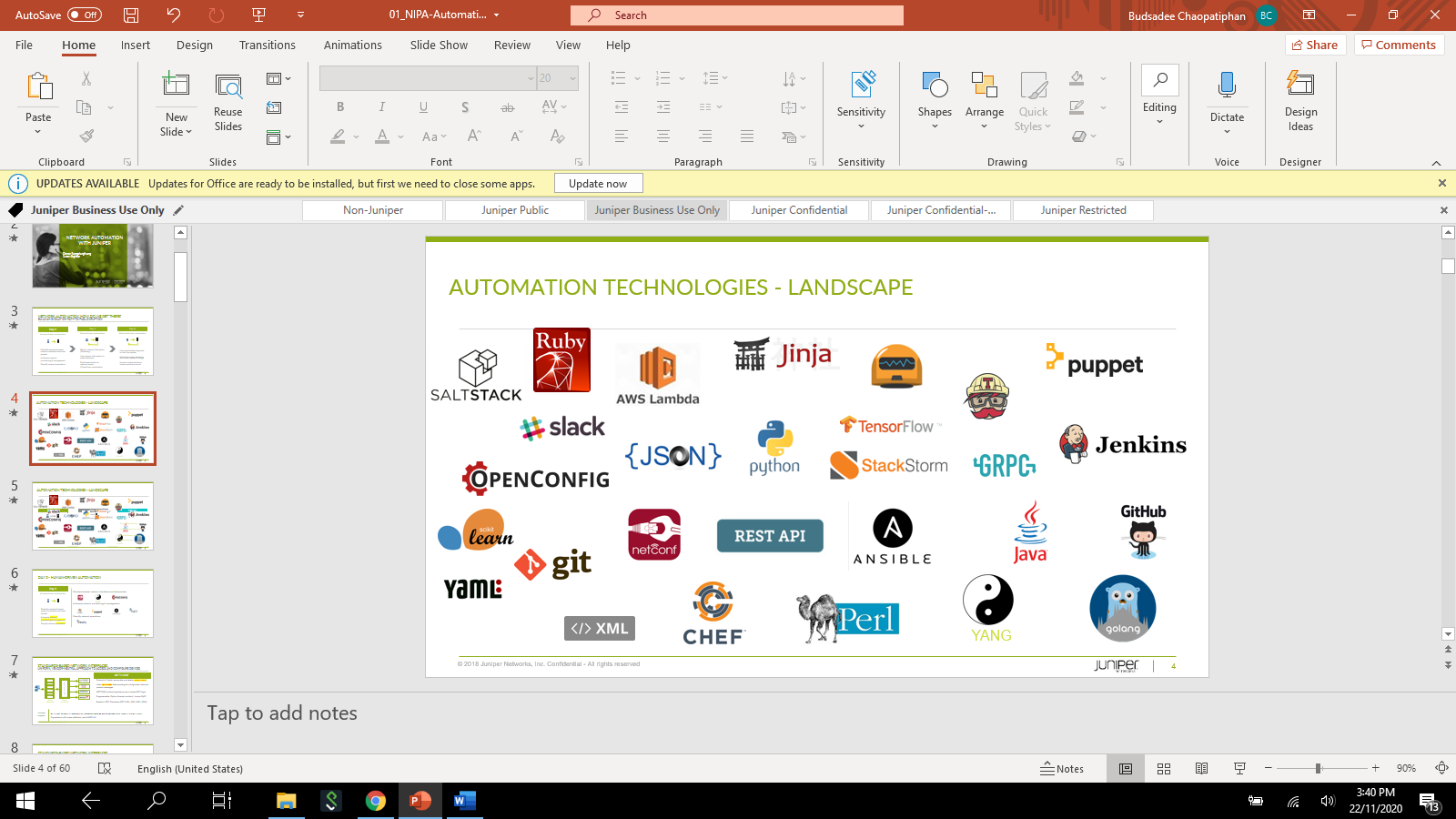
**What is network automation?**

Network automation refers to the automation of network management tasks, including provisioning, configuration, deployment, and security of both physical and virtual network devices. The goal of network automation is maximizing network efficiency and while increasing reliability.

**How does network automation work?**

Network automation finds the most efficient way to map, configure, provision, and manage a network. API-based automation replaces manual, command-line instructions to configure each networking device. The APIs can be invoked directly or go through a programming language, for example Python, Java or Go. Scripts are only one aspect of network automation

**Example of Network Automation Technology**



**Python**

**What is python**

Python is a high-level programming language with dynamic semantics. It uses as a scripting language to connect existing components together also easy to learn.

**How to install and setup Python**

You can download python from this link : <https://www.python.org/downloads/>

Install on Window10 : <https://phoenixnap.com/kb/how-to-install-python-3-windows>

Install on Mac : <https://www.digitalocean.com/community/tutorials/how-to-install-python-3-and-set-up-a-local-programming-environment-on-macos>

Install in Mac OsX : <https://docs.python-guide.org/starting/install3/osx/>

Install on Ubuntu : <https://www.digitalocean.com/community/tutorials/how-to-install-python-3-and-set-up-a-programming-environment-on-an-ubuntu-20-04-server>

**Note :** u can use compile Online to learning python but it can’t install specific library.

**Compile python online** : <https://www.programiz.com/python-programming/online-compiler/>

**How to install module or library**

Installation from PyPI is the most common usage of [pip](https://packaging.python.org/key_projects/#pip) is to install from [Python Package](https://packaging.python.org/glossary/#term-Python-Package-Index-PyPI)

|  |
| --- |
| pip install xxxxx |

To install a specific version:

|  |
| --- |
| pip install xxxxx==1.4 |

Upgrade an already installed some packages to the latest from PyPI.

|  |
| --- |
| pip install --upgrade xxxxx |

**More Information :** <https://packaging.python.org/tutorials/installing-packages/>

**Note** : xxx is refers to name of package such as **junos-eznc , lxml , Jinja2, PyYAML**

**Python Modules for Network Engineer**

Python allows you to import modules to reuse code.

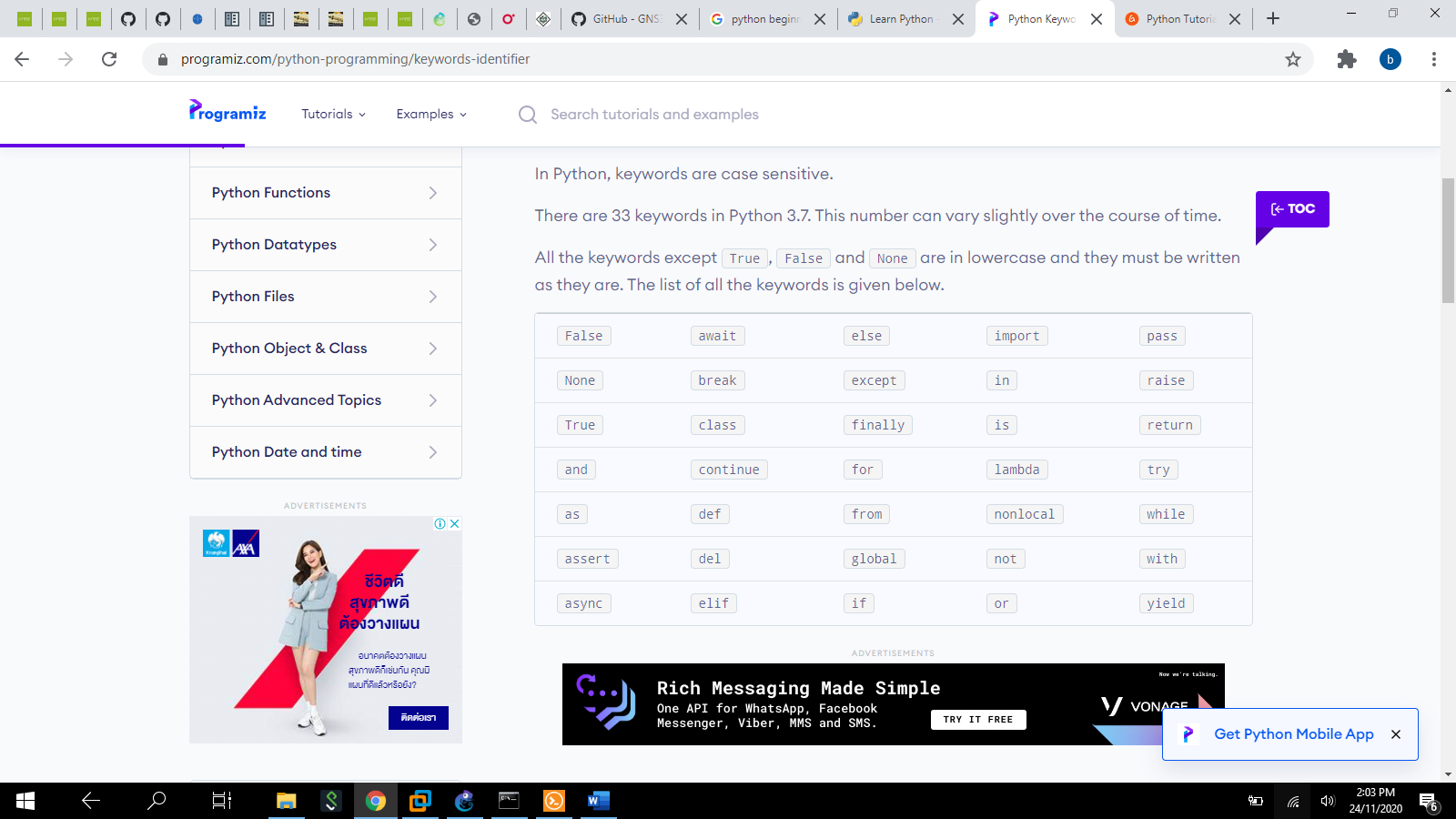
|  |  |
| --- | --- |
| **Library’s name** | **Description** |
| PyEZ | It is a python library to simplifies executing RPCs and performing operational and configuration tasks on devices running Junos OS. |
| lxml | It is a python library to simplifies XPath handling. |
| Jinja2 | It is a python library to generate template engine that enables you to create content from predefined templates, which can be useful for generating Junos OS configuration data. |
| PyYAML | It is a python library for YAML parser and emitter |

**Note** we recommend that you migrate supported script types from Python 2 to Python 3, because support for Python 2.7 might be removed from devices running Junos OS in the future.

**Python Basic**

* **Python Keyword**

We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language. **keywords are case sensitive.**



* **Python Identifiers**

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

**Rules for writing identifiers**

1. Identifiers can be a **combination of letters in lowercase** (a to z) or **uppercase** (A to Z) or **digits** (0 to 9) or an **underscore** \_. Names like myClass, var\_1 and print\_this\_to\_screen, all are valid example.

2. An identifier **cannot start with a digit.** such as,1variable is invalid, but variable1 is a valid name.

3. Keywords cannot **be used as identifiers**.

4. **We cannot use special symbols** like !, @, #, $, % etc.

**Note :** Python is a case-sensitive language. This means, Variable and variable are not the same.

* **Python** **Indentation**

Most of the programming languages like C, C++, and Java use braces **{ }** to define a block of code. **Python uses indentation.**

A code block (body of a function, loop, etc**.) starts with indentation and ends with the first unindent line**. The amount of indentation is up to you, but it must be consistent throughout that block. Generally, **four whitespaces are used for indentation and are preferred over tabs**.

**Here is an example :**

|  |
| --- |
| for i in range(1,11):  print(i)  if i == 5:  break |

Indentation can be ignored in line continuation, but it's always a good idea to indent. It makes the code more readable.

**For example:**

|  |
| --- |
| if True:  print('Hello')  a = 5 |

both are valid and do the same thing, but the former style is clearer.

|  |
| --- |
| if True: print('Hello'); a = 5 |

**Note :** Incorrect indentation will result in **IndentationError.**

* **Python** **Comments & Multi-line comments**

we use the hash (#) symbol for single line comment and multi-line comment, we use triple quotes, either ''' or """

|  |
| --- |
| #This is a comment  #print out Hello  print('Hello') |

|  |
| --- |
| def double(num):  """Function to double the value"""  return 2\*num |

These triple quotes are generally used for multi-line strings. But they can be used as a multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

**Note:** Python Interpreter ignores comments.

* **Python** **Variables**

A variable is a **named location used to store data in the memory**. It is helpful to think of variables as a container that holds data that can be changed later in the program

* you can **use the operator = to assign a value to a variable**.

|  |
| --- |
| website = "apple.com" # string  number = 10 # number |

**Note :** Python is a **type-inferred language**, so you don't have to explicitly define the variable type. It automatically knows what type of variable and declares the value of variable as the same type.

* **variable can change the value of a variable**

|  |
| --- |
| website = "apple.com"  print(website)  # assigning a new value to website  website = "programiz.com"  print(website) |

Output :

|  |
| --- |
| apple.com  programiz.com |

* **Assigning multiple values to multiple variables**

|  |
| --- |
| aa=2,b=3.5,x = y = z = "same"  print (x) # x = same  print (y) # y = same  print (z) # z = same  print (aa) # aa = 2  print (b) # b = 3.5 |

* **Constants**

A constant is a type of variable whose **value cannot be changed**. It **is helpful to** think of constants as containers that **hold information which cannot be changed later.**

* **Literals**

Literal is a raw data given in a variable or constant. In Python, there are various types of literals they are as follows: **Numeric Literals** are immutable (unchangeable).

Numeric literals can belong to 3 different numerical types: Integer, Float, and Complex.

**For example :**

|  |
| --- |
| a = 0b1010 #Binary Literals  b = 100 #Decimal Literal  c = 0o310 #Octal Literal  d = 0x12c #Hexadecimal Literal  #Float Literal  float\_1 = 10.5  float\_2 = 1.5e2  #Complex Literal  x = 3.14j  print(a, b, c, d)  print(float\_1, float\_2)  print(x, x.imag, x.real) |

Output :

|  |
| --- |
| 10 100 200 300  10.5 150.0  3.14j 3.14 0.0 |

* **String literals**

A string literal is a sequence of characters **surrounded by quotes**. We can **use** both **single, double, or triple quotes for a string**. a **character litera**l is a single character surrounded **by single or double quotes.**

|  |
| --- |
| strings = "This is Python"  char = "C"  multiline\_str = """This is a multiline string with more than one line code."""  unicode = u"\u00dcnic\u00f6de"  raw\_str = r"raw \n string"  print(strings)  print(char)  print(multiline\_str)  print(unicode)  print(raw\_str) |

**Output:**

|  |
| --- |
| This is Python  C  This is a multiline string with more than one line code.  Ünicöde  raw \n string |

* **Boolean literals**

A Boolean literal can have any of the two values: **True or False.**

|  |
| --- |
| x = (1 == True)  y = (1 == False)  a = True + 4  b = False + 10  print("x is", x)  print("y is", y)  print("a:", a)  print("b:", b) |

**Output:**

|  |
| --- |
| x is True  y is False  a: 5  b: 10 |

* **Python** **Input, Output and Import**

**print()**

1. We use the **print()** function **to output data to the standard output device (screen).**

|  |
| --- |
| a = 5  print('The value of a is', a) |

**Output:**

|  |
| --- |
| The value of a is 5 |

2. **Output formatting**

Sometimes we would like to format our output to make it look attractive. This can be done by using the **str.format()** method. This method is visible to any string object.

|  |
| --- |
| >>> x = 5; y = 10  >>> print('The value of x is {} and y is {}'.format(x,y))  The value of x is 5 and y is 10  >>> print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))  Hello John, Goodmorning |

**Note :** the curly braces {} are used as placeholders. We can specify the order in which they are printed by using numbers

|  |
| --- |
| print('I love {0} and {1}'.format('bread','butter'))  print('I love {1} and {0}'.format('bread','butter')) |

**Output :**

|  |
| --- |
| I love bread and butter  I love butter and bread |

**Input()**

To allow flexibility, we might **want to take the input from the user**. In Python, we have the input() function to allow this.

|  |
| --- |
| input([prompt]) |

**Note:** where prompt is **the string we wish to display on the screen**.

|  |
| --- |
| >>> num = input('Enter a number: ')  Enter a number: 10  >>> num  '10' |

Here, we can see that the **entered value 10 is a string, not a number**. To **convert this into a number** we can **use int() or float()** functions.

|  |
| --- |
| >>> int('10') #10  >>> float('10') #10.0 |

**Import**

When our program grows bigger, it is a good idea to break it into different modules.

A **module is a file containing Python definitions and statements**. Python modules have a filename and end with the extension **.py :** such as runPy.py

Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import keyword to do this.

|  |
| --- |
| import xxxx |

**Note:** xxxx is refer to name of module such as math, io, sys

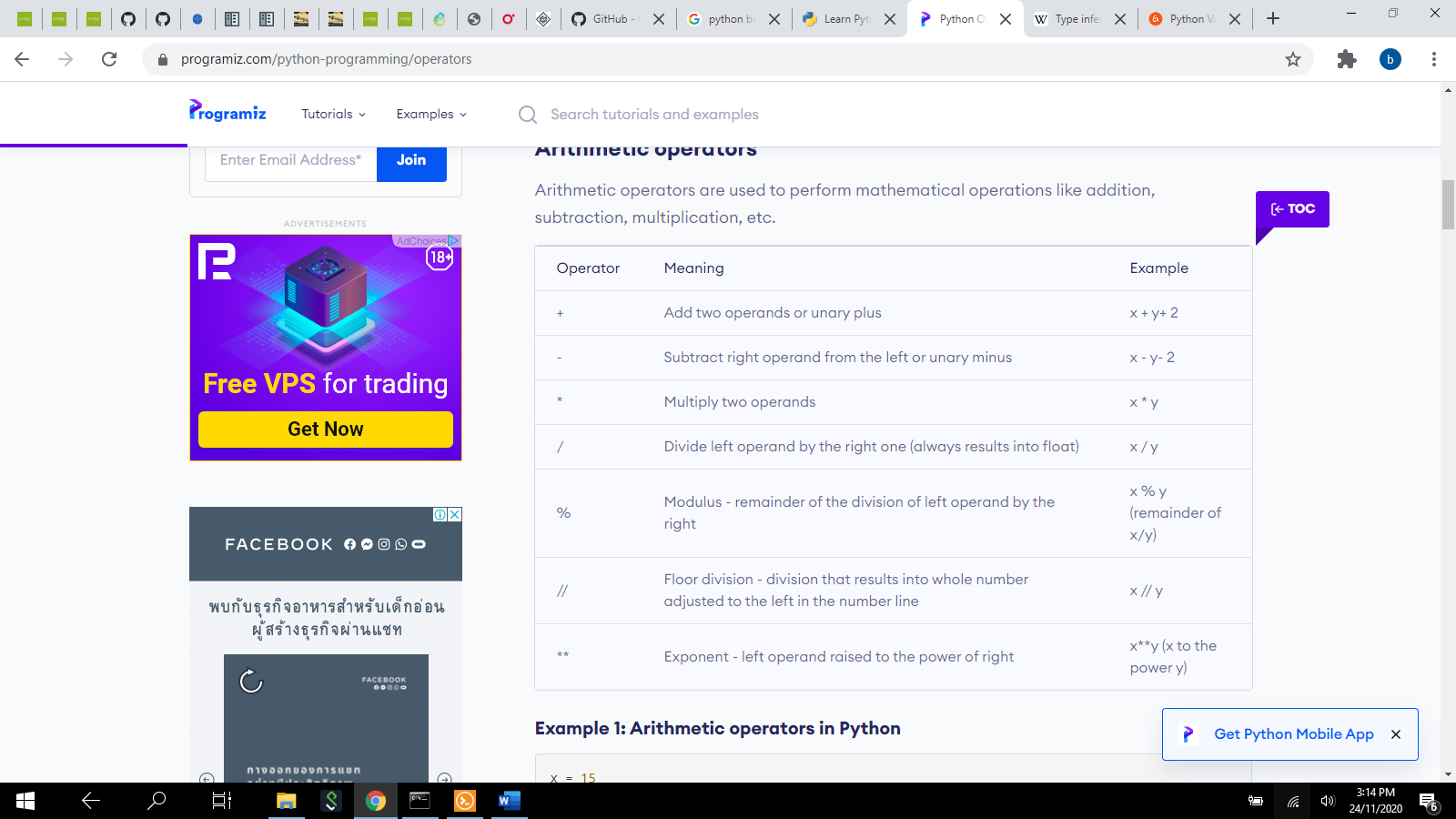
We can also import some specific attributes and functions only, using the from keyword.

|  |
| --- |
| >>> from math import pi  >>> import sys  >>> pi # 3.141592653589793  >>> sys.path |

* **Operators**

**Arithmetic operators**

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

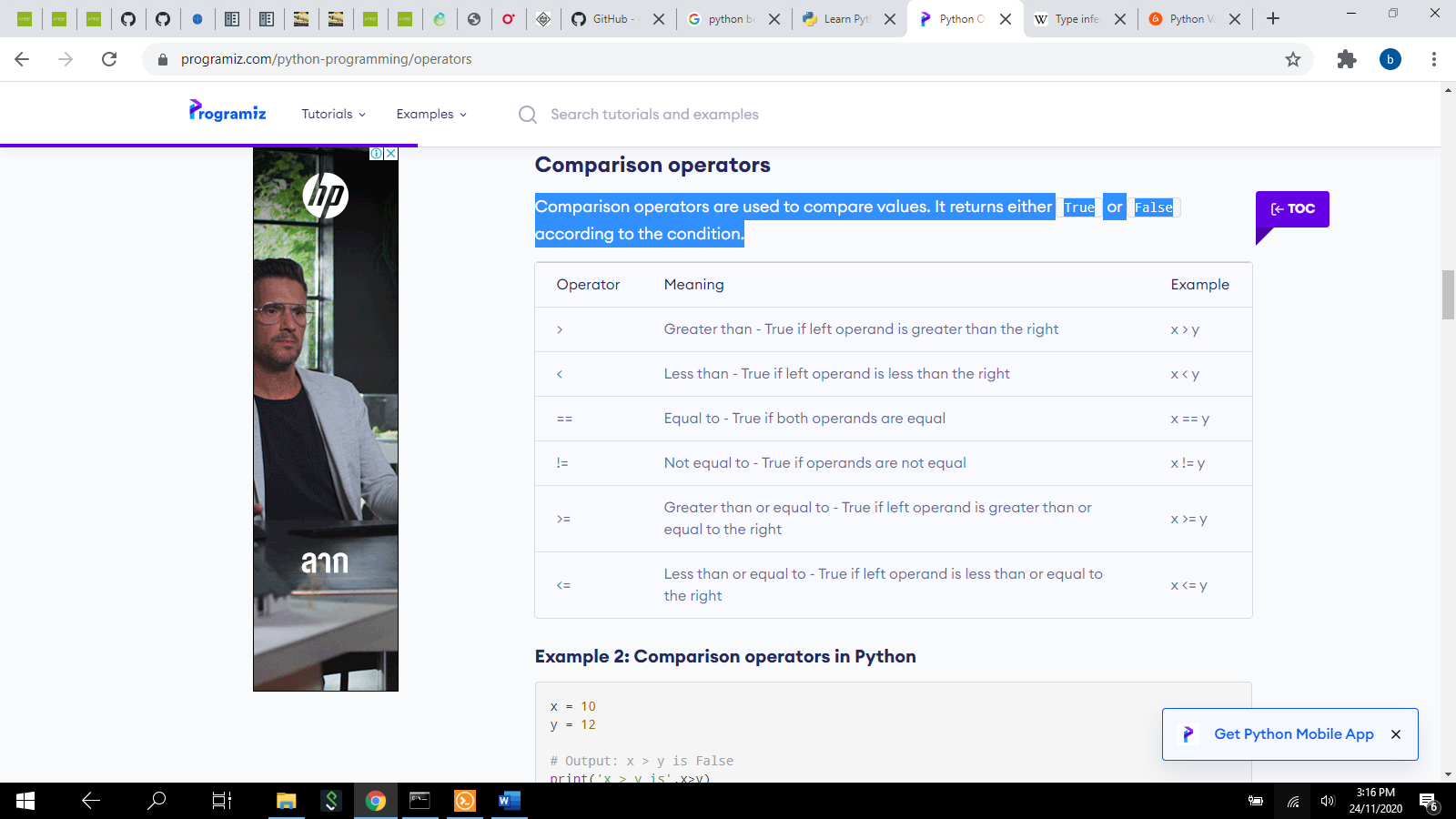


**For example :**

|  |
| --- |
| x = 15  y = 4  print('x + y =',x+y) # Output: x + y = 19  print('x - y =',x-y) # Output: x - y = 11  print('x \* y =',x\*y) # Output: x \* y = 60  print('x / y =',x/y) # Output: x / y = 3.75  print('x // y =',x//y) # Output: x // y = 3  print('x \*\* y =',x\*\*y) # Output: x \*\* y = 50625 |

**Comparison operators**

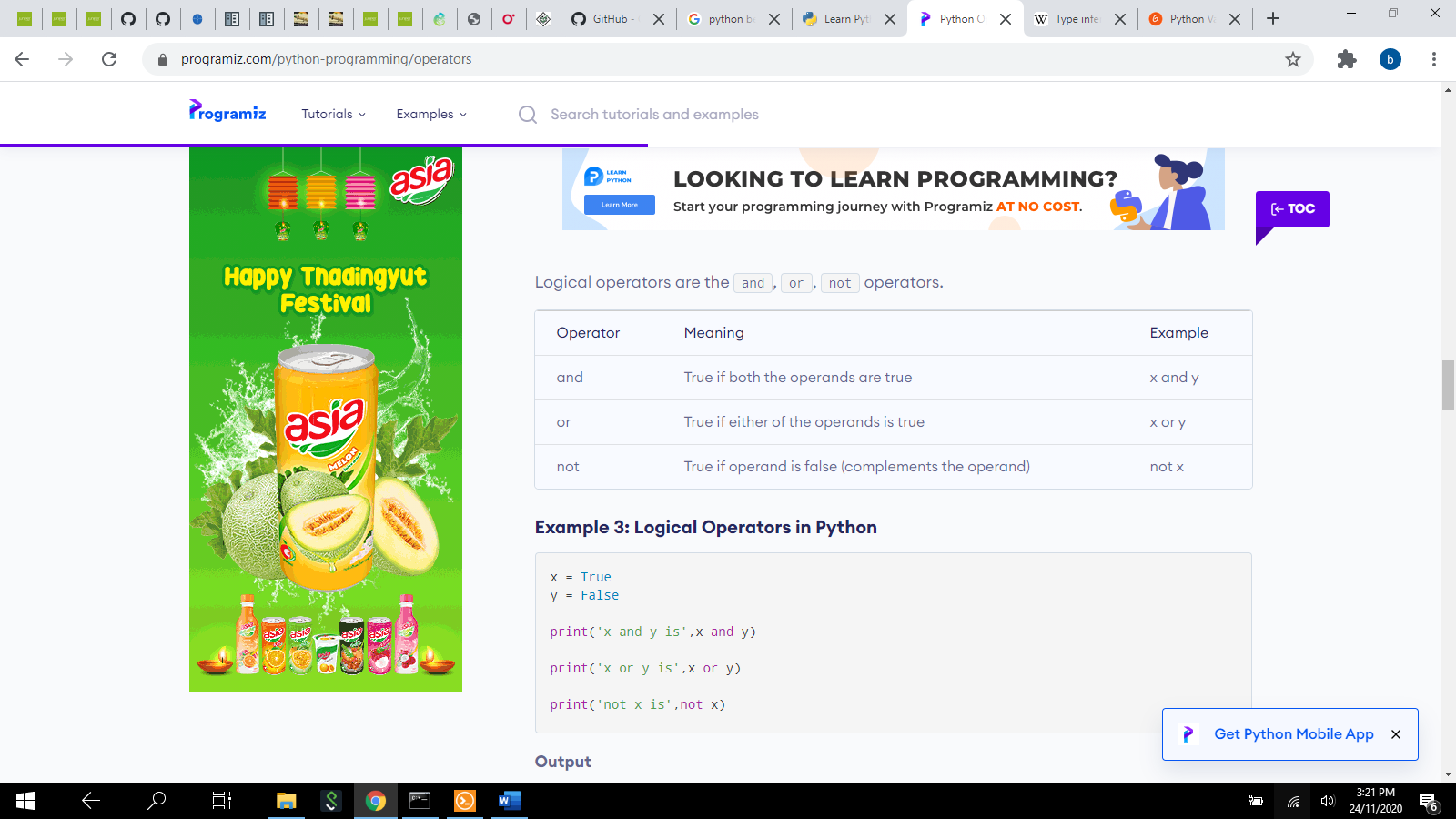
Comparison operators are used to compare values. It **returns either True or False** according to the condition.



**For example :**

|  |
| --- |
| x = 10  y = 12  print('x > y is',x>y) # Output: x > y is False  print('x < y is',x<y) # Output: x < y is True  print('x == y is',x==y) # Output: x == y is False  print('x != y is',x!=y) # Output: x != y is True  print('x >= y is',x>=y) # Output: x >= y is False  print('x <= y is',x<=y) # Output: x <= y is True |

**Logical operators**



**For example :**

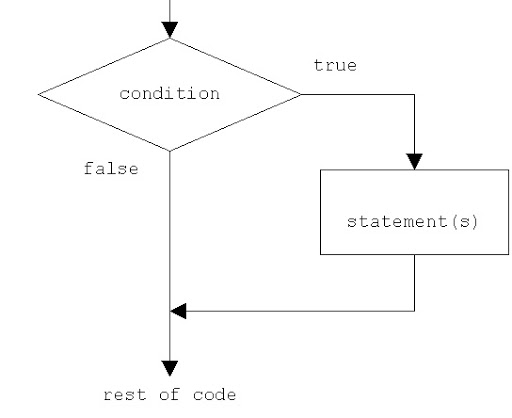
|  |
| --- |
| x = True  y = False  print('x and y is',x and y) # x and y is False  print('x or y is',x or y) # x or y is True  print('not x is',not x) # not x is False |

* **Python** **Flow Control**

**If Syntax**

|  |
| --- |
| if test expression:  statement(s) |

the program evaluates the test expression and **will execute statement(s) only if the test expression is True**. If the **test expression is False, the statement(s) is not executed.**

****

**Note : T**he body of the if statement is indicated by the indentation. The body starts with an indentation and the first unindent line marks the end.

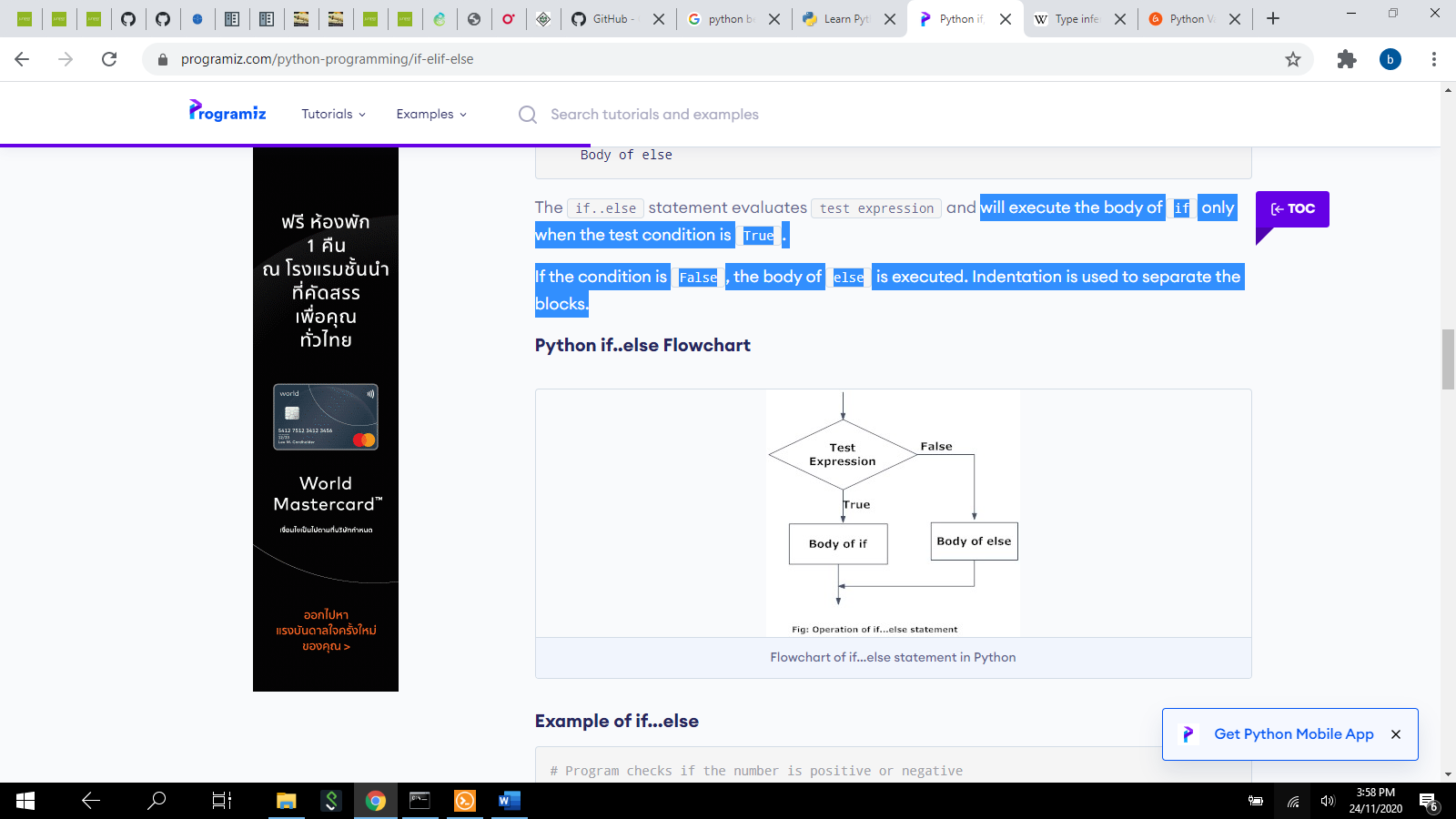
**For example :**

|  |
| --- |
| # If the number is positive, we print an appropriate message  num = 3  if num > 0:  print(num, "is a positive number.")  print("This is always printed.")  num = -1  if num > 0:  print(num, "is a positive number.")  print("This is also always printed.") |

**if...else Syntax**

|  |
| --- |
| if test expression:  Body of if  else:  Body of else |

It will **execute the body of if only when the test condition is True. If the condition is False, the body of else is executed**. **Indentation is used to separate the blocks.**



**For example :**

|  |
| --- |
| num = 3  if num >= 0:  print("Positive or Zero")  else:  print("Negative number") |

**Output:**

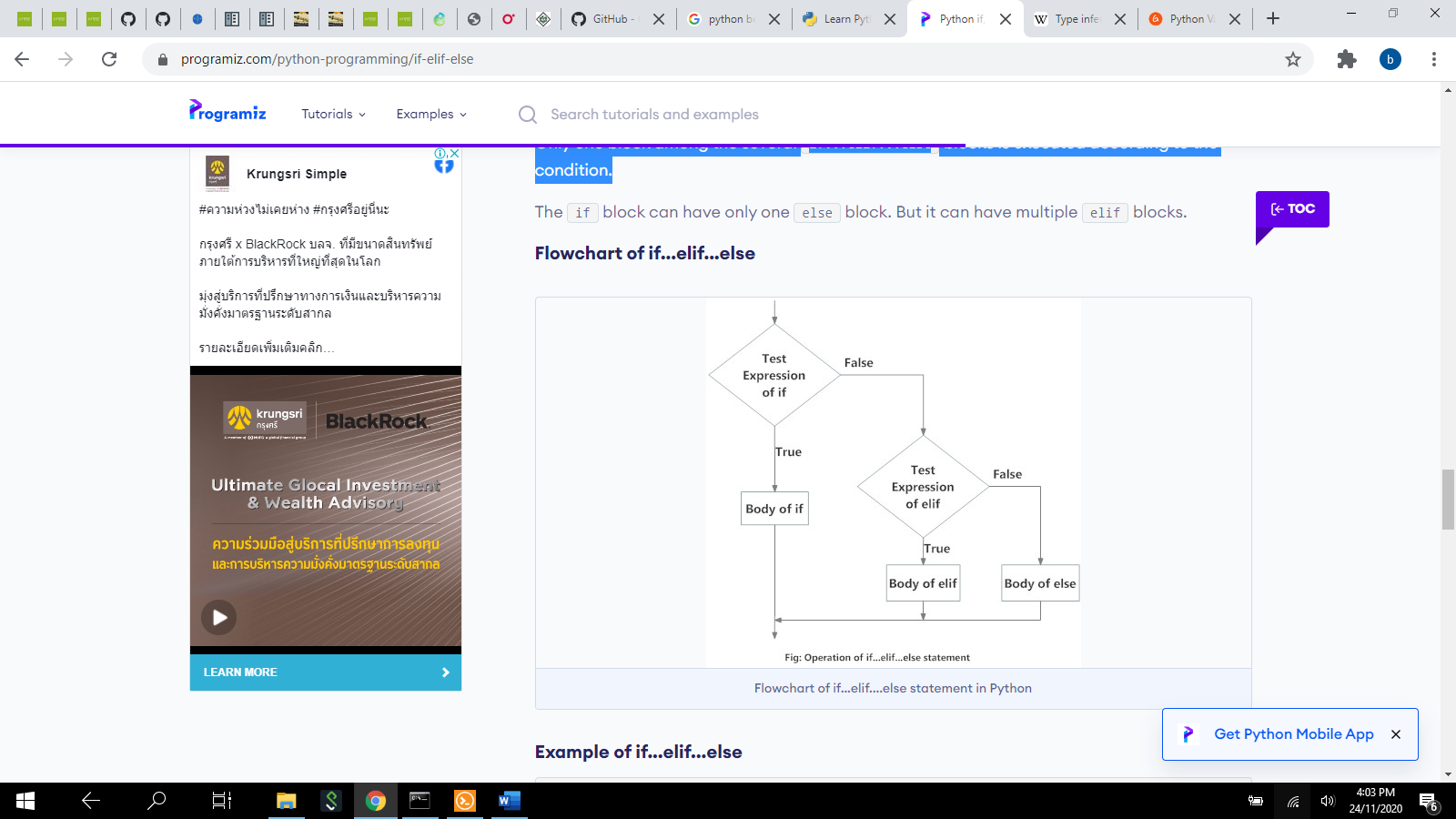
|  |
| --- |
| Positive or Zero |

**if...elif...else Syntax**

|  |
| --- |
| if test expression:  Body of if  elif test expression:  Body of elif  else:  Body of else |

The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, the body of else is executed.

**Note**: Only one block among the several if...elif...else blocks is executed according to the condition.



**For example :**

|  |
| --- |
| num = 3.4 # Try these two variations as well: num = 0 , -4.5  if num > 0:  print("Positive number")  elif num == 0:  print("Zero")  else:  print("Negative number") |

**Python For Loop**

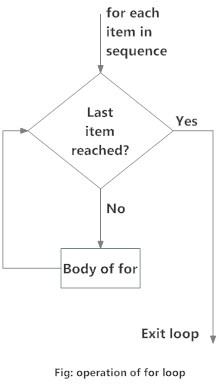
For loop in Python is used to iterate over a sequence (list, tuple, string) or other alterable objects. Iterating over a sequence is called traversal.

**For loop syntax**

|  |
| --- |
| for val in sequence:  Body of for |

val is the variable that takes the value of the item inside the sequence on each iteration.

**Note:** Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.



**For example:**

|  |
| --- |
| # Program to find the sum of all numbers stored in a list  numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11] # List of numbers  sum = 0 # variable to store the sum  # iterate over the list  for val in numbers:  sum = sum+val  print("The sum is", sum) |

**Output:**

|  |
| --- |
| The sum is 48 |

**Python While Loop**

A while loop is repeatedly **executes a target statement as long as a given condition is true.**

|  |
| --- |
| while expression:  statement(s) |

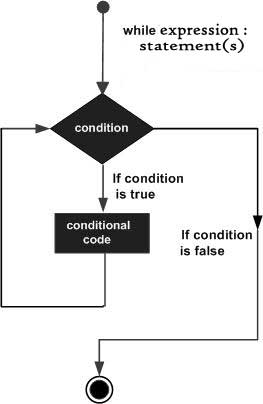
**For example:**

|  |
| --- |
| count = 0  while (count < 5):  print ('The count is:', count)  count = count + 1  print ("Good bye!") |

**Output:**

|  |
| --- |
| The count is: 0  The count is: 1  The count is: 2  The count is: 3  The count is: 4  Good bye! |

**Note:** When the condition becomes false, program control passes to the line immediately following the loop.



**Python Nested Loops**

Python allows to use one loop inside another loop.

**Nested for loops syntax**

|  |
| --- |
| for val in sequence:  for val\_2 in sequence\_2:  statements(s)  statements(s) |

**Nested while loop**

|  |
| --- |
| while expression:  while expression:  statement(s)  statement(s) |

**For example: loop to find the prime numbers from 2 to 20**

|  |
| --- |
| #(i%j==0 is true , other are false)  i = 2  while(i < 20):  j = 2  while(j <= (i/j)):  if not(i%j): break  j = j + 1  if (j > i/j) :  print (i, " is prime")  i = i + 1  print ("Good bye!") |

**Output:**

|  |
| --- |
| 2 is prime  3 is prime  5 is prime  7 is prime  11 is prime  13 is prime  17 is prime  19 is prime  Good bye! |

**range() function**

you can **generate a sequence of numbers using range()** function. range(10) will generate numbers from 0 to 9 (10 numbers) and **range() can define the start, stop and step size.** such as range(start, stop,step\_size).

**Note:** step\_size defaults to 1 if not provided.

**For example :**

|  |
| --- |
| # Program to iterate through a list using indexing  genre = ['pop', 'rock', 'jazz']  #len() => count index of list  for i in range(len(genre)):  print("I like", genre[i]) # i refer to index of list |

**Output :**

|  |
| --- |
| I like pop  I like rock  ​I like jazz |

**Apply for loop with else**

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts. The **break** keyword **can be used to stop a for loop.**

**For example:**

|  |
| --- |
| student\_name = 'Soyuj'  marks = {'James': 90, 'Jules': 55, 'Arthur': 77}  for student in marks:  if student == student\_name:  print(marks[student])  break  else:  print('No entry with that name found.') |

**Output:**

|  |
| --- |
| No entry with that name found. |

**Note:** The break statement **terminates the loop containing it**. Control of the program flows to the statement immediately after the body of the loop. **If the break statement is inside a nested loop** (loop inside another loop), **the break statement will terminate the innermost loop**.

**For example :**

|  |
| --- |
| for val in "string":  if val == "i":  break  print(val)  print("The end") |

**Output :**

|  |
| --- |
| s  t  r  The end |

**Python** **Exception**

exceptions can be handled using a **try…except..else** statement. The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.

**For example I:**

|  |
| --- |
| import sys # import module sys to get the type of exception  randomList = ['a', 0, 2]  for entry in randomList:  try:  print("The entry is", entry)  r = 1/int(entry)  break  except Exception as e:  print("Oops!", sys.exc\_info()[0], "occurred.")  print("Next entry.")  print()  print("The reciprocal of", entry, "is", r) |

**Output:**

|  |
| --- |
| The entry is a  Oops! <class 'ValueError'> occurred.  Next entry.  The entry is 0  Oops! <class 'ZeroDivisionError'> occured.  Next entry.  The entry is 2  The reciprocal of 2 is 0.5 |

**For example II :**

|  |
| --- |
| try:  fh = open("testfile", "w")  fh.write("This is my test file for exception handling!!")  except IOError:  print ("Error: can\'t find file or read data")  else:  print ("Written content in the file successfully")  fh.close() |

**Output:**

|  |
| --- |
| Error: can't find file or read data |

**More Information** : <https://www.tutorialspoint.com/python/python_exceptions.htm>

**Python Function / Method**

A function is a block of organized, reusable code.

**Rule to define a Function**

1. Function blocks begin with the keyword **def** followed by the function name and parentheses **( ).**

2. Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

3. The first statement of a function can be an optional statement

4. The code block within every function starts with a colon ( **:** ) and is indented.

5. The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

**Function Syntax:**

|  |
| --- |
| def functionname( parameters ):  statement  return [expression] |

**Note :** Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

**Calling a Function**

you can execute function **by calling it from another function or directly from the Python prompt**.

**For example:**

|  |
| --- |
| def printme( str ):  print (str)  return;  # Now you can call printme function  printme("I'm first call to user defined function!")  printme("Again second call to the same function")  printme( str = "My name is tutu") |

**Output:**

|  |
| --- |
| I'm first call to user defined function!  Again second call to the same function  My name is tutu |

**More Information :** <https://www.tutorialspoint.com/python/python_functions.htm>

**Python Class**

**Classes** are used to create new local namespace. where all its attributes are defined. Attributes may be data (object) or functions (method , which identify the behaviors and actions).

Like function definitions begin with the **def** keyword in Python, class definitions begin with a **class** keyword.

**Class Syntax :**

|  |
| --- |
| class nameOfClass:  statement |

**\_\_init\_\_() Function**

This special function gets called whenever a new object of that class is instantiated. It’s called **constructor method**.

**Note:** Use the \_\_init\_\_() function to assign values to object properties, or other operations that are necessary to do when the object is being created

**For example:**

|  |
| --- |
| class Person:  def \_\_init\_\_(**self**, name, age):  self.name = name  self.age = age  p1 = Person("John", 36)  print(p1.name)  print(p1.age) |

**Output:**

|  |
| --- |
| John  36 |

**Note:** The **self parameter** is a reference to the **current instance of the class**, and is used to **access variables that belong to the class**. It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class

**For example II :**

|  |
| --- |
| #Use the words mysillyobject and abc instead of self:  class Person:  def \_\_init\_\_(mysillyobject, name, age):  mysillyobject.name = name  mysillyobject.age = age  def myfunc(abc):  print("Hello my name is " + abc.name)  p1 = Person("John", 36)  p1.myfunc() |

**Output:**

|  |
| --- |
| Hello my name is John |

**For example III:** Instance variables are for data unique to each instance and class variables are for attributes and methods

|  |
| --- |
| class Dog:  animal = 'dog' # Class Variable  # The init method or constructor  def \_\_init\_\_(self, breed, color):  self.breed = breed # Instance Variable  self.color = color  # Adds an instance variable  def setType(self, type):  self.type = type    # Retrieves instance variable  def getType(self):  return self.type |

**Calling Class :**

|  |
| --- |
| Rodger = Dog("Pug", "brown") # 1. Create Objects of Dog class  Buzo = Dog("Bulldog", "black")  Rodger.setType("Dog From American")  print('Rodger details:')  print('Rodger is a', Rodger.animal)  print('Breed: ', Rodger.breed)  print('Color: ', Rodger.color)    print('\nBuzo details:')  print('Buzo is a', Buzo.animal)  print('Breed: ', Buzo.breed)  print('Color: ', Buzo.color)    print("\nAccessing class variable using class name")  print(Dog.animal) # Class variables can be accessed using class  print(Rodger.getType()) |

Output :

|  |
| --- |
| Rodger details:  Rodger is a dog  Breed: Pug  Color: brown  Buzo details:  Buzo is a dog  Breed: Bulldog  Color: black  Accessing class variable using class name  Dog  Dog From American |

**Junos PyEZ Overview**

**What is it?**

Junos PyEZ is a **microframework for Python** that enables you to manage and automate devices running Junos OS using the familiarity of Python. It is **designed to provide the capabilities that a user would have on the Junos OS command-line interface (CLI) in an environment built for automation tasks.** Junos PyEZ **does not require extensive knowledge of Junos OS or the Junos XML APIs.**

However, you **do not have to be an experienced programmer to use Junos PyEZ**. Non-programmers can quickly execute simple commands in Python interactive mode, and more experienced programmers can opt to create more complex, robust, and reusable programs to perform tasks.

**Junos PyEZ configuration enable you to:**

* retrieve configuration data
* compare configurations
* upload and commit configuration changes
* roll back the configuration
* manage the rescue configuration

Junos PyEZ consists of the **jnpr.junos** package, which contains modules that handle device connectivity and provide operational and configuration utilities.

**Junos’s module :**

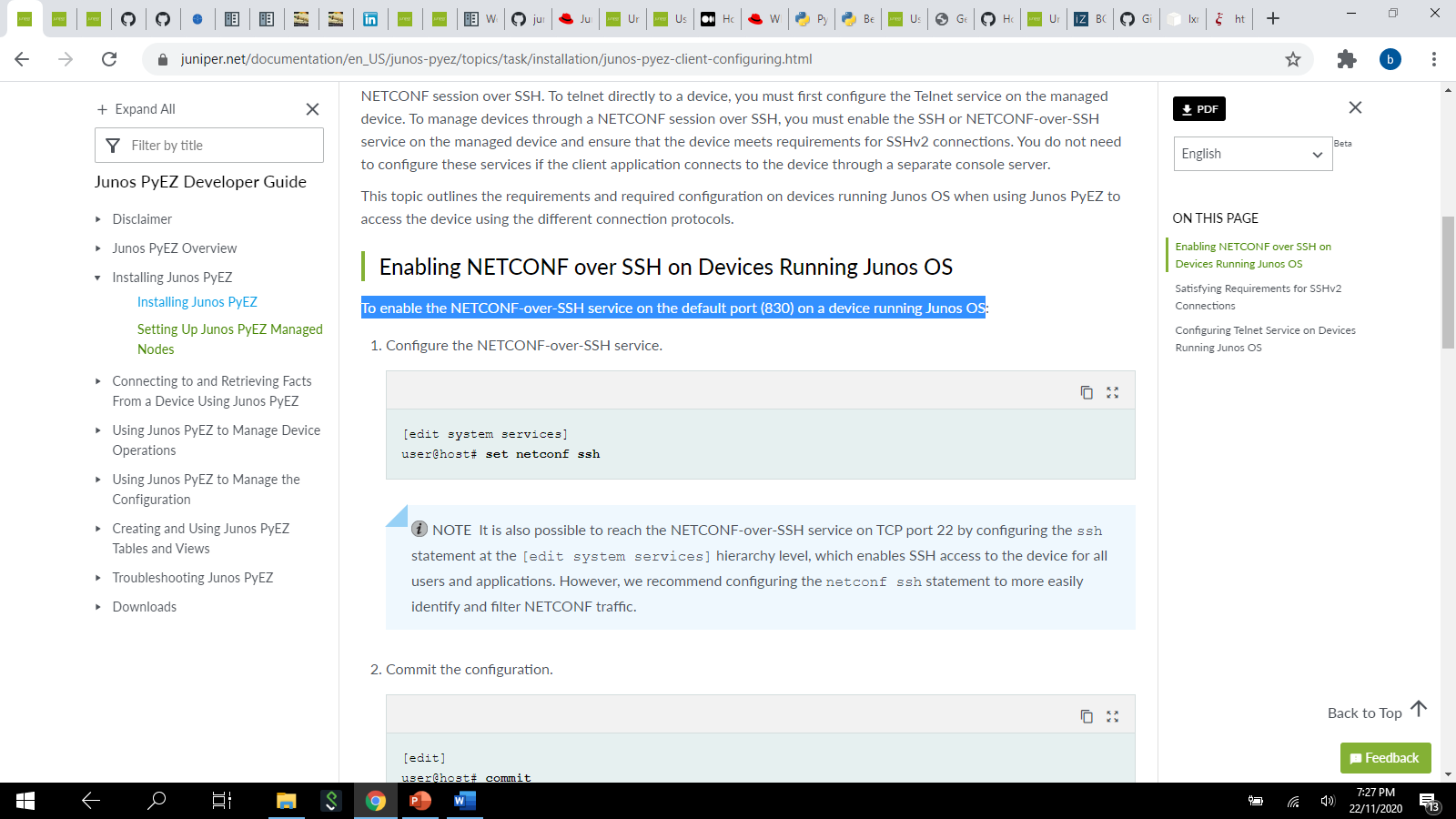
|  |  |
| --- | --- |
| **Module’s name** | **Description** |
| device | Defines the Device class, which represents the device running Junos OS and enables you to connect to and retrieve facts from the device. |
| command | Includes predefined operational Tables and Views that can be used to filter unstructured output returned from CLI and vty commands and convert it to JSON. |
| exception | Defines exceptions encountered when managing devices running Junos OS. |
| factory | Contains code pertaining to Tables and Views, including the **loadyaml()** function, which is used to load custom Tables and Views. |
| facts | A dictionary-like object of read-only facts about the device. These facts are accessed using the facts attribute of a Device object instance. |
| op | Includes predefined operational Tables and Views that can be used to filter structured output returned from RPCs. |
| resources | Includes predefined configuration Tables and Views representing specific configuration resources, which can be used to programmatically configure devices running Junos OS. |
| transport | Contains code used by the Device class to support the different connection types. |
| utils | Defines submodules and classes that handle software installation, file system and copy operations, and configuration management. |

**Setting Up Junos PyEZ Managed Nodes**

To manage devices through a NETCONF session over SSH, you must enable the SSH or NETCONF-over-SSH service on the managed device and ensure that the device meets requirements for SSHv2 connections. You do not need to configure these services if the client application connects to the device through a separate console server.

**Enabling NETCONF over SSH on Devices Running Junos OS**

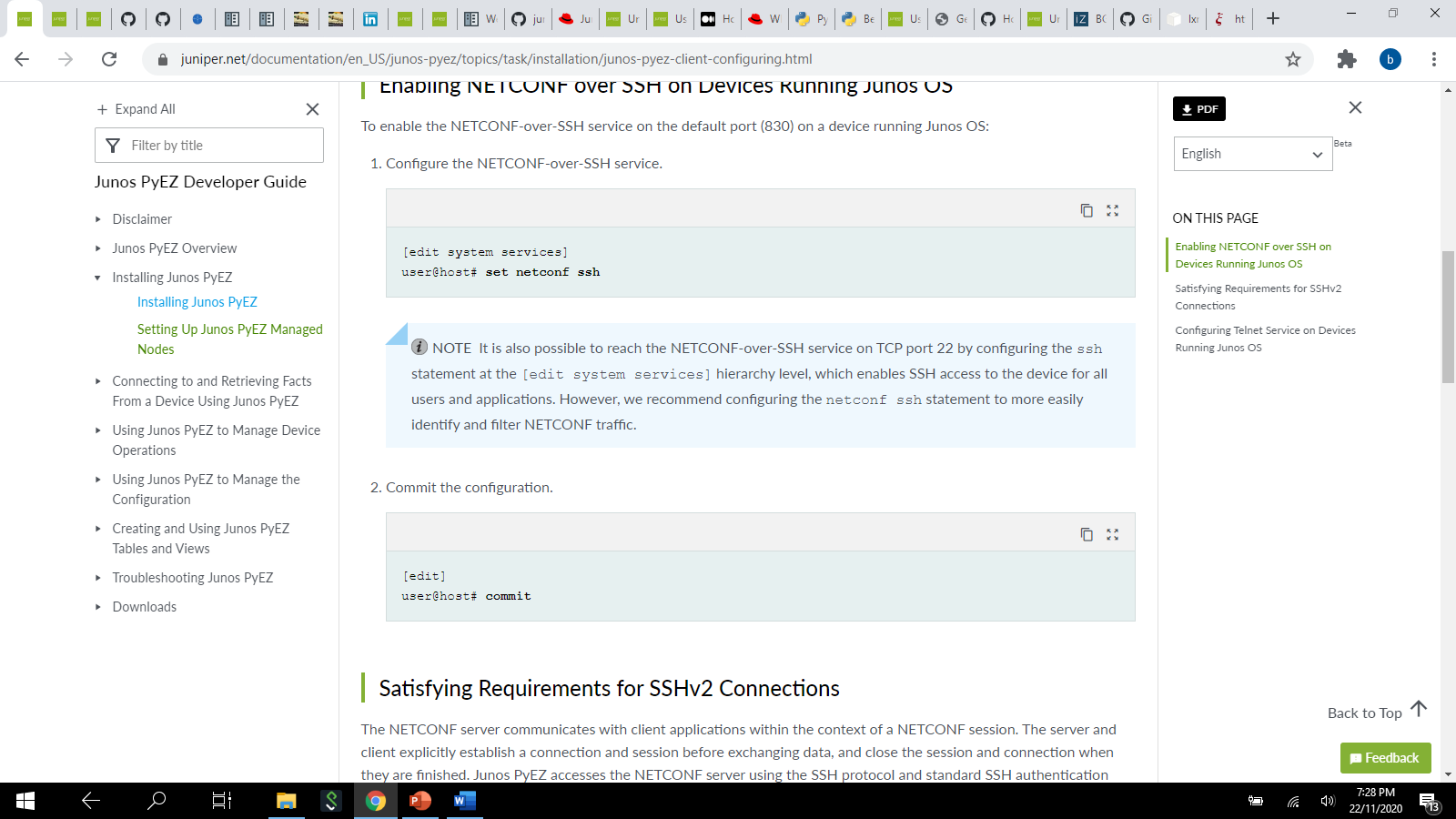
To **enable the NETCONF-over-SSH service on the default port (830)** on a device running Junos OS



Or use this command:

|  |
| --- |
| **set system services netconf ssh** |

**Commit** the configuration



**Connect to devices, retrieve facts with JUNOS PyEZ**

Junos PyEZ models each device as an instance of the **jnpr.junos.device**. The **Device** class enables you **to connect to a device running Junos OS using a serial console connection, telnet, or by establishing a NETCONF session over SSH**.

New or zeroized devices that have factory default configurations require access through a console connection. Thus, you can use Junos PyEZ to initially configure a device that is not yet configured for remote access by using either a serial console connection when you are directly connected to the device or by using telnet or SSH through a console server that is connected to the device. By default, Junos PyEZ uses SSH to connect to a device.

**Understanding Junos PyEZ Connection Properties**

When you connect to a device running Junos OS, Junos PyEZ stores information about the current connection as properties of the Device instance. so, you must understand connection properties.

|  |  |
| --- | --- |
| **Properties’ name** | **Description** |
| connected | Boolean specifying the current state of the connection.  Returns **True** when connected. |
| hostname | String specifying the hostname of the device to which the application is connected. |
| master | Boolean returning True if the Routing Engine to which the application is connected is the primary Routing Engine. |
| port | Integer or string specifying the port used for the connection. |
| re\_name | String specifying the Routing Engine name to which the application is connected. |
| timeout | Integer specifying the RPC timeout value in seconds. |
| uptime | Integer representing the number of seconds since the current Routing Engine was booted. This property is available starting in Junos PyEZ Release 2.1.5. |
| user | String specifying the user accessing the device running Junos OS. |

**For example I : Connect A device**

Step1 : Import device class from package **jnpr.junos**

|  |
| --- |
| **from** **jnpr.junos** **import** Device |

Step2 : Instantiate the class device by declaring an instance of the class calling and passing argument to device class

|  |
| --- |
| dev = Device(host="172.30.111.111", user="lab", password="lab123") |

|  |
| --- |
| type(dev) |

**Note** : You can show type of variable with **type** command

Step3 : Use the method **open** to connect to the device

|  |
| --- |
| dev.open() |

Note : Now you can get properties define in the class device. Example,

|  |
| --- |
| >>> dev.connected  True |
| >>> dev.user  'lab' |
| >>> dev.timeout  30 |

Step4 : Close Connection

|  |
| --- |
| dev.close() |

**For example II :**

Step1 : Import Device class and any other modules or objects required for your tasks.

|  |
| --- |
| **import** **sys**  **from** **getpass** **import** getpass  **from** **jnpr.junos** **import** Device  **from** **jnpr.junos.exception** **import** ConnectError  **from** **pprint** **import** pprint |

Step2 : Create the device instance, and provide the hostname, any parameters required for authentication, and any optional parameters.

|  |
| --- |
| hostname = input("Device hostname: ")  junos\_username = input("Junos OS username: ")  junos\_password = getpass("Junos OS **or** SSH key password: ")  dev = Device(host=hostname, user=junos\_username, passwd=junos\_password) |

Step3 : Connect to the device by calling the open() method, for example:

|  |
| --- |
| **try**:  dev.open()  **except** ConnectError **as** err:  **print** ("Cannot connect to device: {0}".format(err))  sys.exit(1)  **except** **Exception** **as** err:  **print** (err)  sys.exit(1) |

Steps 4 : Print the device facts.

Facts is a property defined in the class Device. It is a dictionary. This command returns the facts. (retrieved facts when the connection is established).

|  |
| --- |
| **pprint** (dev.facts) |

This command, you can move it after you use open command

|  |
| --- |
| **try**:  dev.open()  **pprint** (dev.facts) |

Steps5 : After performing any necessary tasks, close the connection to the device.

|  |
| --- |
| dev.close() |

Note: Junos PyEZ automatically queries the default SSH configuration file at ~/.ssh/config, if one exists. However, you can specify a different SSH configuration file when you create the device instance by including the **ssh\_config** parameter in the Device argument list.

**For example II.I**

|  |
| --- |
| ssh\_config\_file = "~/.ssh/config\_dc"  dev = Device(host='198.51.100.1', ssh\_config=ssh\_config\_file) |

**For example III : Connecting to a Device Using Telnet**

|  |
| --- |
| **import** **sys**  **from** **getpass** **import** getpass  **from** **jnpr.junos** **import** Device  hostname = input("Device hostname: ")  junos\_username = input("Junos OS username: ")  junos\_password = getpass("Junos OS password: ")  **try**:  **with** Device(host=hostname, user=junos\_username, passwd=junos\_password, mode='telnet', port='23') **as** dev:  **print** (dev.facts)  **except** **Exception** **as** err:  **print** (err)  sys.exit(1) |

**For example IV : Authenticating Junos PyEZ Users Using a Password**

Steps 1 : Include code that prompts for the hostname to which to connect and the username and password for the device running Junos OS and stores each value in a variable.

|  |
| --- |
| **from** **jnpr.junos** **import** Device  **from** **getpass** **import** getpass  **import** **sys**  hostname = input("Hostname: ")  junos\_username = input("Junos OS username: ")  junos\_password = getpass("Junos OS password: ") |

Steps2 : If the Junos PyEZ client connects to the device through an SSH connection to a console server, include code that prompts for the console server username and password and stores each value in a variable.

|  |
| --- |
| cs\_username = input("Console server username: ")  cs\_password = getpass("Console server password: ") |

Step3 : In the Device constructor argument list

|  |
| --- |
| **try**:  *# NETCONF session over SSH*  **with** Device(host=hostname, user=junos\_username, passwd=junos\_password) **as** dev: |
| *# Serial console connection to device*  *#with Device(host=hostname, user=junos\_username, passwd=junos\_password, mode='serial', port='/dev/ttyUSB0') as dev:* |
| *# SSH connection to console server connected to device*  *#with Device(host=hostname, user=junos\_username, passwd=junos\_password, cs\_user=cs\_username, cs\_passwd=cs\_password, timeout=5) as dev:* |
| *# Telnet connection to device or console server connected to device*  *#with Device(host=hostname, user=junos\_username, passwd=junos\_password, mode='telnet', port='23') as dev:* |
| **print** (dev.facts)  **except** **Exception** **as** err:  **print** (err)  sys.exit(1) |

NOTE: All platforms running Junos OS have only the root user configured by default, without any password. When using Junos PyEZ to initially configure a new or zeroized device through a console connection, use user='root', and omit the passwd parameter.

**Using Junos PyEZ to Retrieve Facts**

After connecting to a device running Junos OS, Junos PyEZ applications can retrieve facts about the device. The device facts are accessed as the facts attribute of the Device object.

**For example I**

|  |
| --- |
| **from** **jnpr.junos** **import** Device  **from** **pprint** **import** pprint  **with** Device(host='router1.example.net') **as** dev:  pprint (dev.facts['hostname'])  pprint (dev.facts) |

Note You can see type of facts with this command:

|  |
| --- |
| >>> Type (dev.facts)  <type 'dict'> |

Select some device facts :

|  |
| --- |
| >>> dev.facts["hostname"]  'LAB-EX-VC-Backbone' |
| >>> dev.facts["version"]  '12.3R6.6' |
| >>> dev.serialnumber  ‘JN10302DAFB’ |
| >>> dev.facts["version"]=="14.1R1.2"  False |

Try This :

|  |
| --- |
| pprint (dev.facts.key()) |
| pprint (dev.facts.values()) |

**Config Utility to Configure Devices Running Junos OS**

how to use the **jnpr.junos.utils.config** utility to make unstructured configuration which consist of static or templatized configuration data that is formatted as ASCII text, Junos XML elements, Junos OS set commands, or JavaScript Object Notation (JSON). The Config utility also enables you to roll back to a previously committed configuration or revert to a rescue configuration.

**Configuration Process Overview**

After successfully connecting to a device running Junos OS, to configure the device using the Config utility, first create a Config object and associate it with the Device instance.

**For example I**

|  |
| --- |
| **from** **jnpr.junos** **import** Device  **from** **jnpr.junos.utils.config** **import** Config  dev = Device(host='dc1a.example.com').open()  cu = Config(dev) |

Note : if you want to know type of config, you use

|  |
| --- |
| >>> type (cu)  <class 'jnpr.junos.utils.config.Config'> |

**Methods defined in the class config:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| lock() | Lock the configuration |
| load() | Call **load()** when loading a new complete configuration or modifying specific portions of the configuration |
| rollback() | Call **rollback()** to revert to a previously committed configuration |
| rescue() | Call **rescue()** to load the rescue configuration |
| commit() | Commit the configuration |
| unlock() | Unlock the configuration |
| pdiff() | Retrieve the difference between the candidate configuration and a previously committed configuration |

**For example II**

|  |
| --- |
| **from** **jnpr.junos** **import** Device  **from** **jnpr.junos.utils.config** **import** Config  dev = Device(host='router1.example.com').open()  **with** Config(dev, mode='exclusive') **as** cu:  cu.load(path='configs/junos-config-mx.conf', merge=True)  cu.pdiff()  cu.commit()  dev.close() |

Note: To retrieve the difference between the configurations as an object for further manipulation, call the **diff()** method instead of the **pdiff()** method, and store the output in a variable. For example:

|  |
| --- |
| diff = cu.diff(rb\_id=2)  **print** (diff) |

**Loading Configuration Data from a Local or Remote File**

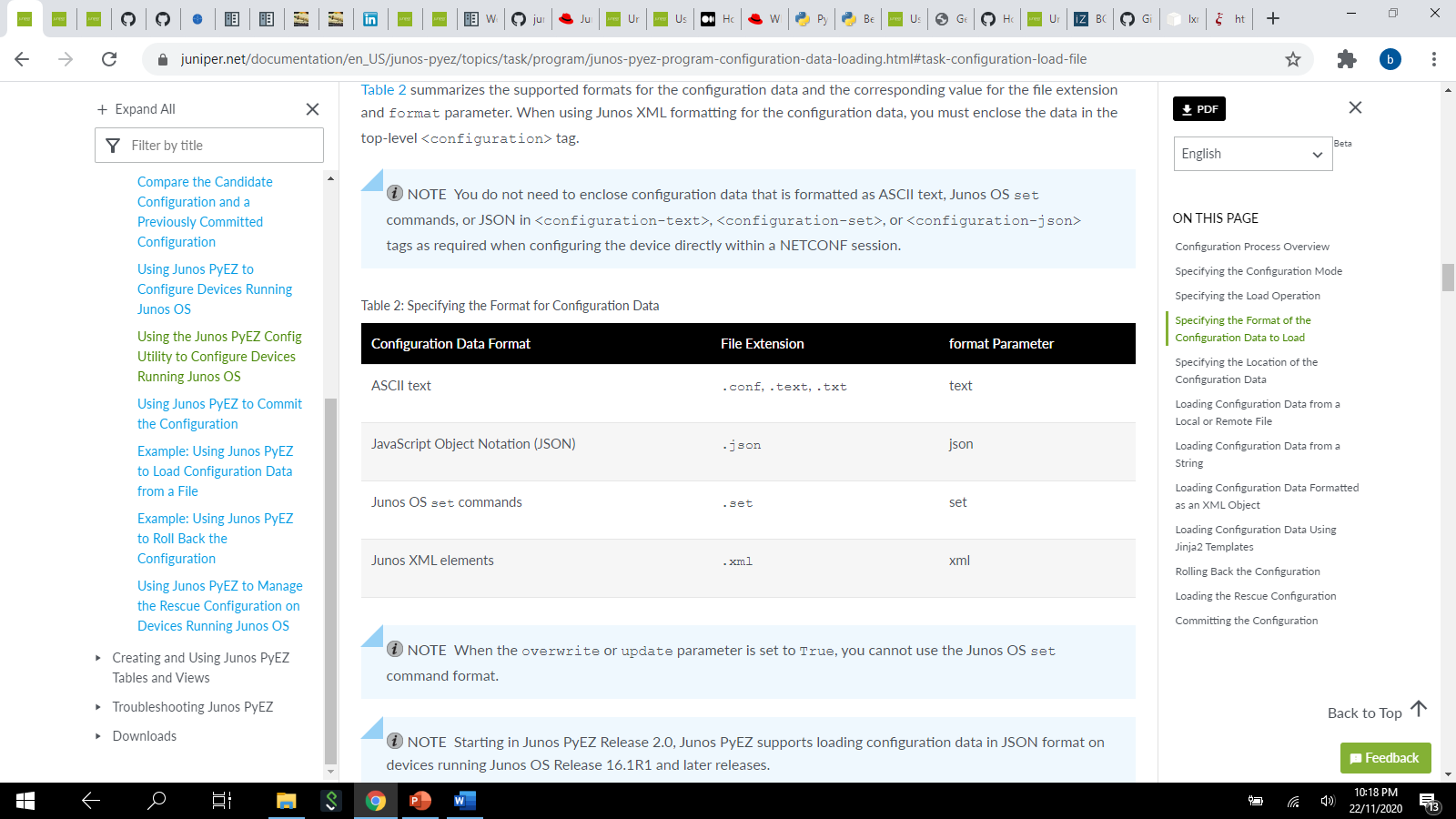
To load configuration data from a local file on the configuration management server, set the **load()** method’s path parameter to the absolute or relative path of the file.

**For example III**

|  |
| --- |
| **from** **jnpr.junos** **import** Device  **from** **jnpr.junos.utils.config** **import** Config  dev = Device(host='dc1a.example.com').open()  conf\_file = 'configs/junos-config-interfaces.conf'  **with** Config(dev, mode='exclusive') **as** cu:  cu.load(**path=conf\_file**, merge=True)  cu.commit()  dev.close() |

To load configuration data from a file on the device running Junos OS or at a remote URL, set the url parameter to the absolute or relative path of the file on the client device or the FTP location or Hypertext Transfer Protocol (HTTP) URL of a remote file, and include any other parameters required for the load operation

|  |
| --- |
| cu.load(url="/var/home/user/golden.conf") |
| cu.load(url="ftp://username@ftp.hostname.net/path/filename") |
| cu.load(url="http://username:password@example.com/path/filename") |



For configuration data formatted as Junos XML:

|  |
| --- |
| config\_xml = """  <configuration>  <system>  <scripts>  <op>  <file>  <name>test.slax</name>  </file>  </op>  </scripts>  </system>  </configuration>  """ |

Load the configuration data by supplying the string as the first argument in the list, and optionally specify **format="xml"**

|  |
| --- |
| cu.load(config\_xml, format="xml", merge=True) |

For configuration data formatted as ASCII text:

|  |
| --- |
| config\_text = """  system {  scripts {  op {  file test.slax;  }  }  }  """ |

Load the configuration data by supplying the string as the first argument in the list, and optionally specify **format="text"**

|  |
| --- |
| cu.load(config\_text, format="text", merge=True) |

For configuration data formatted using JSON:

|  |
| --- |
| config\_json = """{  "configuration" : {  "system" : {  "scripts" : {  "op" : {  "file" : [  {  "name" : "test.slax"  }  ]  }  }  }  }  }""" |

Load the configuration data by supplying the string as the first argument in the list, and optionally specify **format="json"**

|  |
| --- |
| cu.load(config\_json, format="json", merge=True) |

**For example IV**

|  |
| --- |
| **from** **jnpr.junos** **import** Device  **from** **jnpr.junos.utils.config** **import** Config  **from** **lxml.builder** **import** E  config\_xml\_obj = (  E.configuration( *# create an Element called "configuration"*  E.system(  E.scripts(  E.op (  E.file (  E.name("test.slax"),  )  )  )  )  )  )  dev = Device(host='dc1a.example.com').open()  **with** Config(dev, mode='exclusive') **as** cu:  cu.load(config\_xml\_obj, merge=True)  cu.commit()  dev.close() |

**Rolling Back the Configuration**

To roll back the configuration, call the **rollback()** method and set the ***rb\_id*** argument to the ID of the rollback configuration. Valid ID values are 0.

The number of stored previous configurations (maximum is 49)

**For example I**

|  |
| --- |
| **from** **jnpr.junos** **import** Device  **from** **jnpr.junos.utils.config** **import** Config  rollback\_id = int(input("Rollback ID of the configuration to restore: "))  dev = Device(host='dc1a.example.com').open()  **with** Config(dev, mode='exclusive') **as** cu:  cu.rollback(rb\_id=rollback\_id)  cu.pdiff()  cu.commit()  dev.close() |

**Loading the Rescue Configuration**

After creating an instance of the Config class, you use the **rescue()** method to mange the rescue configuration. You specify the action to perform on the rescue configuration by setting the rescue() method action parameter to the desired operation.

**For example I**

|  |
| --- |
| **from** **jnpr.junos** **import** Device  **from** **jnpr.junos.utils.config** **import** Config  dev = Device(host='dc1a.example.com').open()  **with** Config(dev, mode='exclusive') **as** cu:  rescue = cu.rescue(action="reload")  **if** rescue **is** False:  **print** ("No existing rescue configuration.")  **else**:  cu.pdiff()  cu.commit()  dev.close() |

To load the existing rescue configuration into the candidate configuration, specify **action="reload"** **If no rescue configuration exists, the load operation returns False**. After loading the rescue configuration, you must commit the configuration to make it the active configuration on the device.

**Committing the Configuration**

After modifying the configuration, you must commit the configuration to make it the active configuration on the device by calling commit() method.

|  |
| --- |
| **from** **jnpr.junos** **import** Device  **from** **jnpr.junos.utils.config** **import** Config  dev = Device(host='dc1a.example.com').open()  conf\_file = "configs/junos-config-interfaces.conf"  **with** Config(dev, mode='exclusive') **as** cu:  cu.load(path=conf\_file, merge=True)  **cu.commit()**  dev.close() |

To verify the syntax of the configuration without committing it, call the **commit\_check()** method in place of the commit() method.

|  |
| --- |
| cu.commit\_check() |

**Specifying Commit Options**

* Commit Comment

When you commit the configuration, you can include a brief comment to describe the purpose of the committed changes. the *comment* parameter and a message string in the **commit()**

|  |
| --- |
| cu.commit(comment='Configuring ge-0/0/0 interface') |

* Commit Confirm

To require that a commit operation be confirmed **within a specified amount of time after the initial commit**, include the confirm=minutes argument in the commit() or set() method argument list, as appropriate.

|  |
| --- |
| cu.commit(confirm=15) |

You can also specify confirm=True to use the default rollback time of 10 minutes

|  |
| --- |
| cu.commit(confirm=True) |

**Note:** If the commit is not confirmed within the given time limit, the configuration automatically rolls back to the previously committed configuration.

The confirmed commit operation is useful for verifying that a configuration change works correctly and does not prevent management access to the device. If the change prevents access or causes other errors, the automatic rollback to the previous configuration enables access to the device after the rollback deadline passes.