Programming and Python for Network Engineer

Muhammad Yasir Shamim
APNIC Community Trainer

Hafsa Talat
Software Engineer

Agenda

Session 1:

- Tedious Networking Operations
- Network Operational Problems
- Evolution Drift to DevOps
- Programming to rescue
- Why learn Programming
- Programming in Networks
- Orchestration vs Automation
- Automation A Smart way out
- Types of Languages
- Languages for Automation

Session 2:

- Which programming language should we learn
- Why Python
- What is Python
- Python and other languages
- Getting started with Python
- Installing Python
- Python programming modes
- Deeper into python

Session 3:

- Network Integration with Python
- GNS3 DEMO
- Telnet
- Netmiko
- Napalm
- Use Case
- Thinking Process
- Python Code
- Python for Network Engineers
- Closing remarks

Tedious Networking Operations

- Vendor dependent command line syntax and structures
- Follow Standard Operating Procedures for tasks
- Low level operational works
- Daily chores drain out technical ability of an Engineer
- Maintenance of manual databases of many non-tech entities
- Learning theoretical network concepts to perform troubleshooting of seldom issues

Network Operational Problems

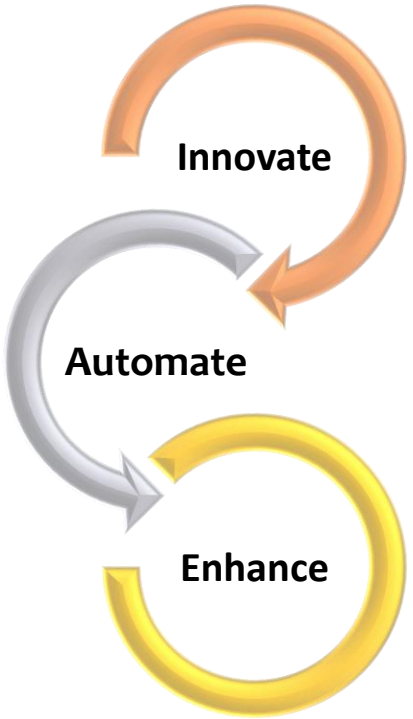
- Daily repetitive tasks
- Highly time consuming operations
- Fat fingers and mistakes
- Infrastructure scaling
- Difficulty of understanding complex scenarios by less technical team members

Evolutional Drift to DevOps

- Software-defined networks using centralized controllers
- Introduction of GUI and replacing CLI legacy
- Inclusion of DevOps is changing networking infrastructure rapidly
- Era of cloud computing and mobility, businesses are demanding more agility from their IT professionals
- Addition of programming skillset in bucket

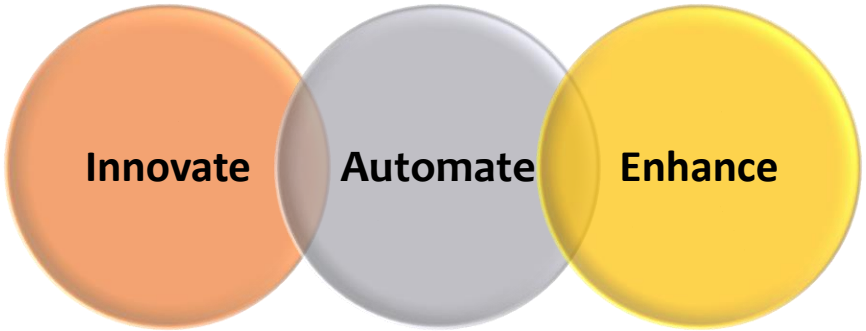
Programming to the Rescue

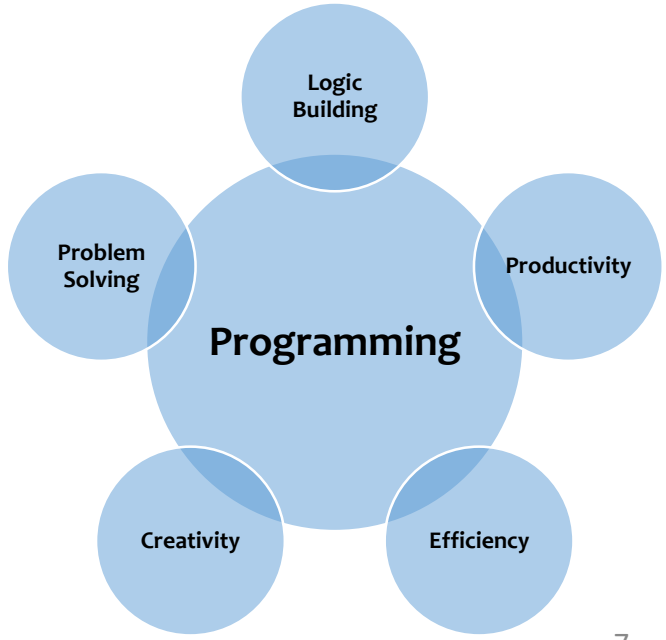
- Networker Engineers no longer need to learn all the different syntax from diverse vendors, codes can do that !
- Predefined code, eliminates the need to follow long configuration steps
- Scalability of networks, adding new operational nodes to the networks in just few clicks.
- Zero tolerance to humanistic errors
- Routine network operations made easy for all
- One window solution for all operational tasks



Why learn Programming

- If software eats everything, are network engineers on the menu?
- An Essential Skill For Network Engineers to enhance his capabilities.
- Programming to simplify or automate tasks.
- Programming languages are not just for programmers. If you are a network engineer, knowing a programming language (or two or three) can come in handy.





Programming in Networks

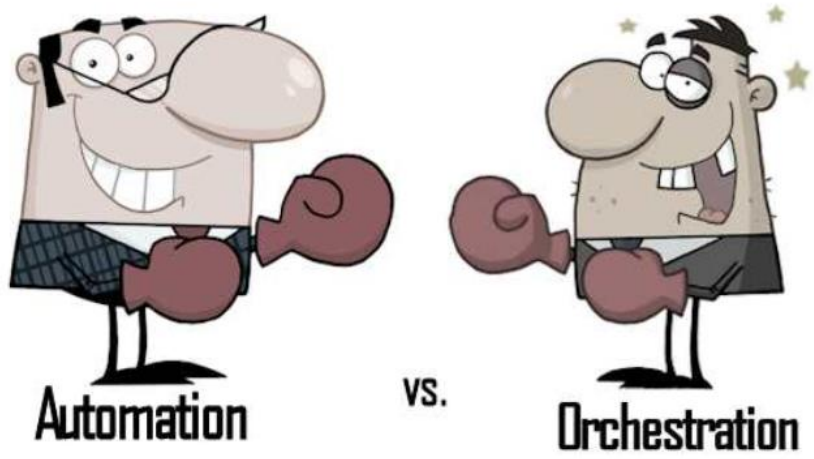
- Manages network more efficiently.
- Network Automation.
- Software define networking.
- Big vendors heading towards software based operations i.e. Cisco, Juniper.

"Your network is a crime scene, and you are the detective. You need better ways to investigate what happened, and prove guilt or innocence".

-- Jeremy

Orchestration vs Automation

- Automation
  - Well specified task run on its own
- Orchestration
  - Automating a lot of things at once
  - Multiple tasks to execute a workflow
  - Is automation, coordination and management
- Automation is the first step towards orchestration



Orchestration vs Automation

Scenario – Network Engineer needs to configure Customer link

What information Network Engineer is looking for:

- Last mile configuration (Metro Fiber)
  - Free switch interface
  - Free vlan
- Configuring the Router
  - Interface (vlan)
  - IP required (Public/Private)
  - Bandwidth

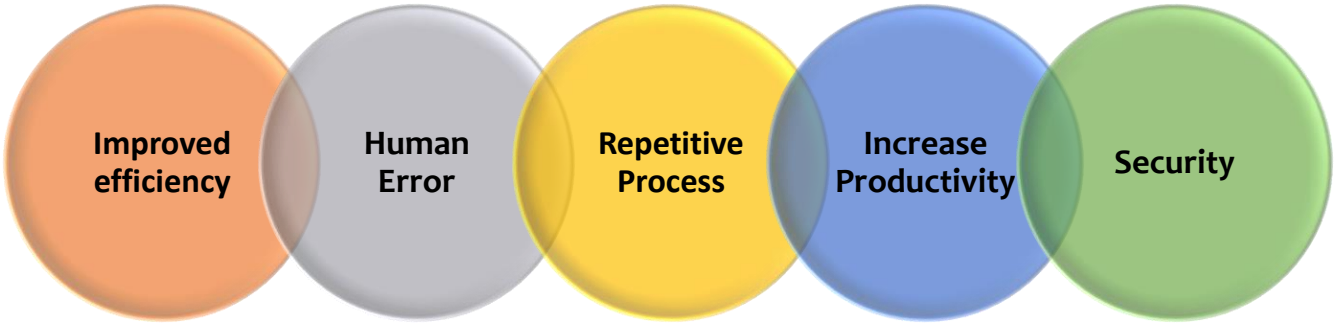
Each individual tasks can be automated, and the whole process is orchestrated

“Automation” A Smart Way Out !

- **Network Automation** is a methodology in which programmed scripts automatically configures, provisions, manages and tests network devices

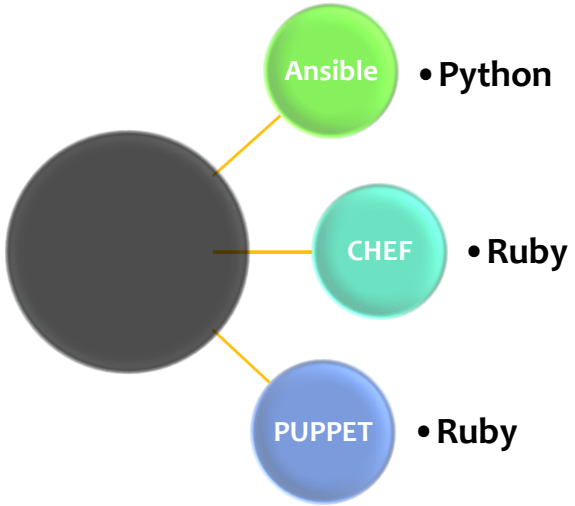
- **Why Automation?** - Programmed management tools are designed to reduce the complexity of manually configuring and managing distributed infrastructure resources by enabling speed, ensuring reliability and compliance

- Using different **programming languages** provide different paths to achieve a common goal of managing infrastructure efficiently



Languages For Automation

- Strong binding between Network Automation Tools and Programming languages.
- Puppet, Chef and Ansible are different paths to achieve a common goal of managing infrastructure efficiently.
- All these configuration management tools are designed to reduce the complexity of configuring distributed infrastructure resources, enabling speed, ensuring reliability and compliance.



CHEF™ ANSIBLE puppet

Which Programming Language should we learn

Easy Learning

User Friendly

Technical Online Resources

Size of Community

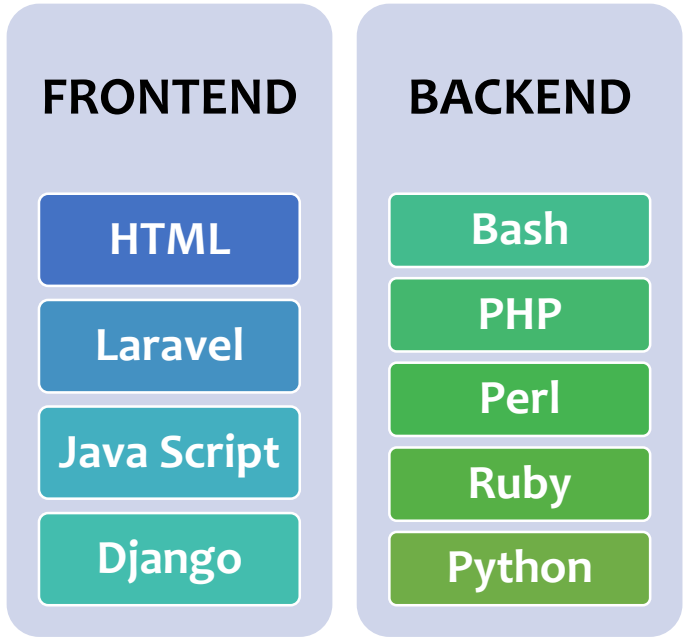
What People are available

Market demands !

Which Language ?

Type of Languages

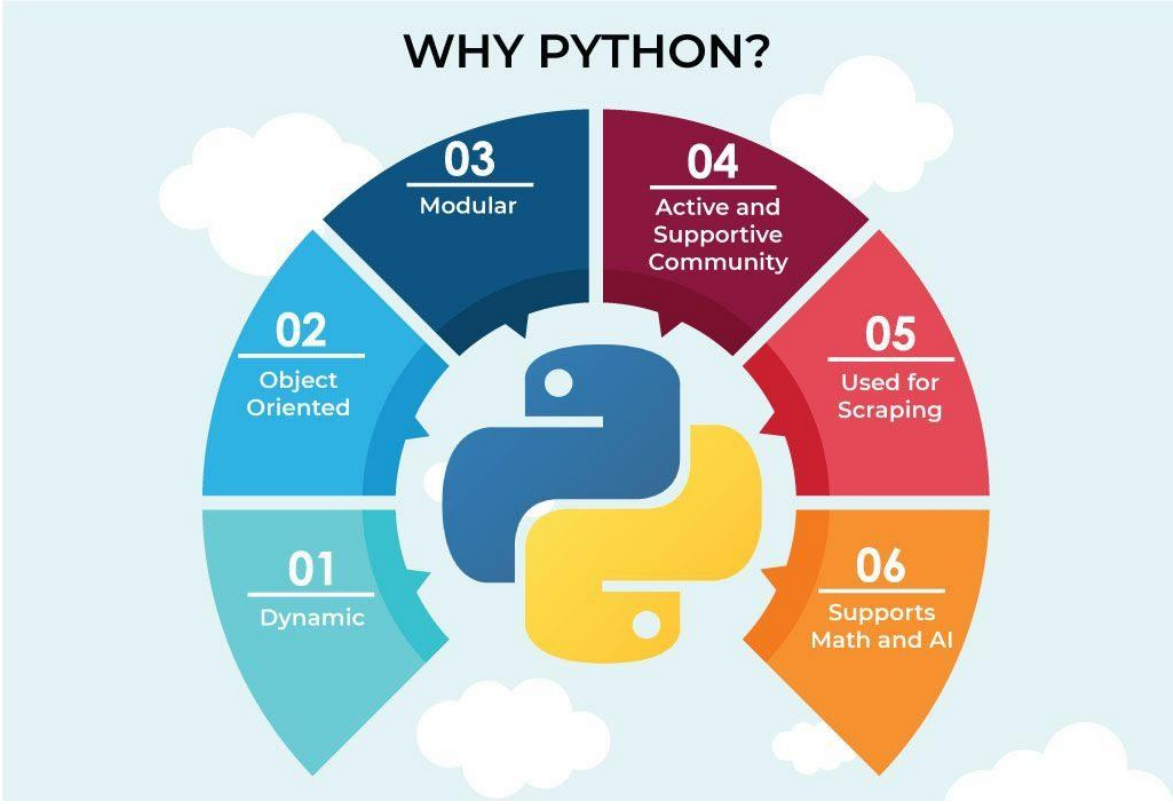
- **Frontend** development is the practice of converting data to graphical interface for user to view and interact with data.
- The digital interaction is accomplished by using HTML, CSS and JavaScript.
- **Backend** of an application is responsible for things like calculations, logic, database interactions, and performance.
- If we talk about networks, it is used to interact with Nodes and perform actions such as run commands, gather stats, perform configurational changes and view output.



What is Python

- Python is a high-level programming language
- Open source and community driven
- “Batteries Included”
  - a standard distribution includes many modules
- Dynamic typed
- Source can be compiled or run just-in-time
- Similar to perl, tcl, ruby

Why Python


WHY PYTHON?
03
Modular
04
Active and Supportive Community
02
Object Oriented
05
Used for Scraping
01
Dynamic
06
Supports Math and AI

Why Python

- Easy to understand and readable language.
- Dominating language at this point of time in Network Automation space.
- A high level language, don't have to write a lot of blue codes to get things done.
- Powerful enough to be used as a convenient tool for daily parsing tasks, performance management, and configuration.
- It does not have to be compiled, which makes debugging really fast and easy

Why Python

- **Interpreted**
  - You run the program straight from the source code.
  - Python program →Bytecode →a platforms native language
  - You can just copy over your code to another system and it will auto-magically work! *with python platform
- **Object-Oriented**
  - Simple and additionally supports procedural programming
- **Extensible** – easily import other code
- **Embeddable** –easily place your code in non-python programs
- **Extensive libraries**
  - (i.e. reg. expressions, doc generation, CGI, ftp, web browsers, ZIP, WAV, cryptography, etc…) (wxPython, Twisted, Python Imaging library)

Why Python

- **Simple**
  - Python is a simple and minimalistic language in nature
  - Reading a **good** python program should be like reading English
  - Its Pseudo-code nature allows one to concentrate on the problem rather than the language
- **Easy to Learn**
- **Free & Open source**
  - Freely distributed and Open source
  - Maintained by the Python community
- **High Level Language** –memory management
- **Portable** – *runs on anything c code will

Python and Other Languages

- **Python is Code-Friendly**
  The code syntax of Python are simple, concise, and much like English language. This makes coding an interactive and engaging activity.

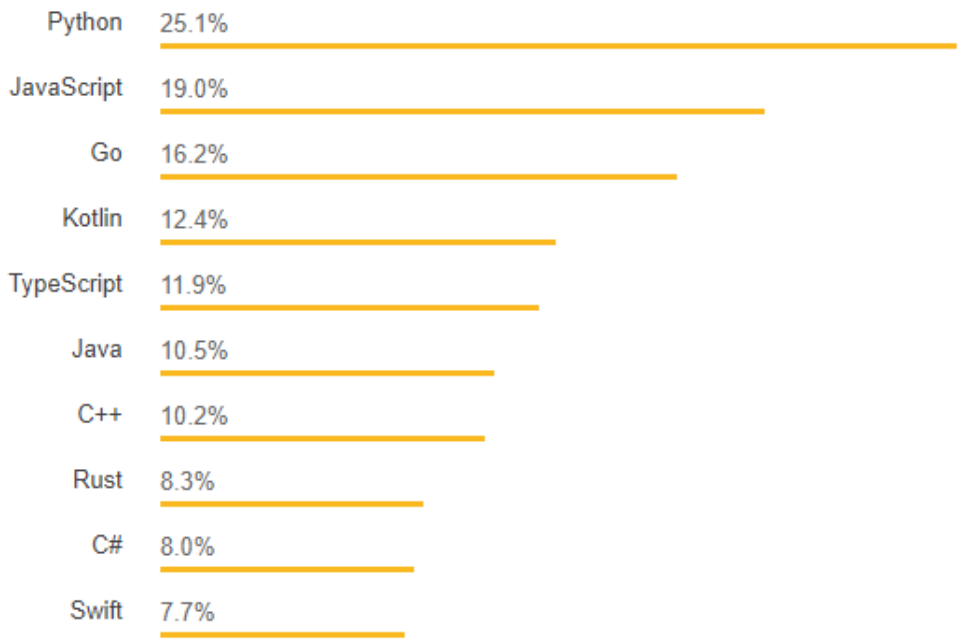
- **It is Procedural and Object-Oriented Programming Language**
  Another best thing about Python is that it follows both the Procedure-oriented and Object-Oriented concepts. Because of this, it is able to provide developers with an environment that no other language concentrating on a single concept is able to offer.

- **It comes with Plenty of Libraries and Frameworks**
  Python, unlike many other programming languages, avails a wide range of libraries and frameworks that gives developers an escape from writing code from scratch. And eventually, lower down the time and cost of development.

- **Python Enjoys Growing Market Popularity**
  Above all, Python is one of the trending programming languages with the support of a wider community. Also, its popularity graph as per Google Trends is growing astronomically - something that is making developers choose Python over other available options.

Python and Other Languages

- Since 2012, Python has been consistently growing in popularity, and the trend is likely to continue, if not increase, in the future



Getting started with Python

Usually a system may have one of the 3 famous Operating System.

- Microsoft Windows
- Apple Mac OS
- Linux Distribution OS

Python development environment is not included in Windows default setup. For **Windows 10,** a small patch does the trick where as for earlier versions proper setup is required. Coding could be done on CMD window.

In **Mac OS X,** Python 2 and its built-in **Interactive Development Environment (IDE)** comes pre-installed. Everything could be done in Terminal that includes creating the code and its execution.

Famous **Linux Distributions** also have Python enabled on default setup. Just as Mac OS, a user can start coding and execute it in Linux terminal window using CLI.

Installing Python

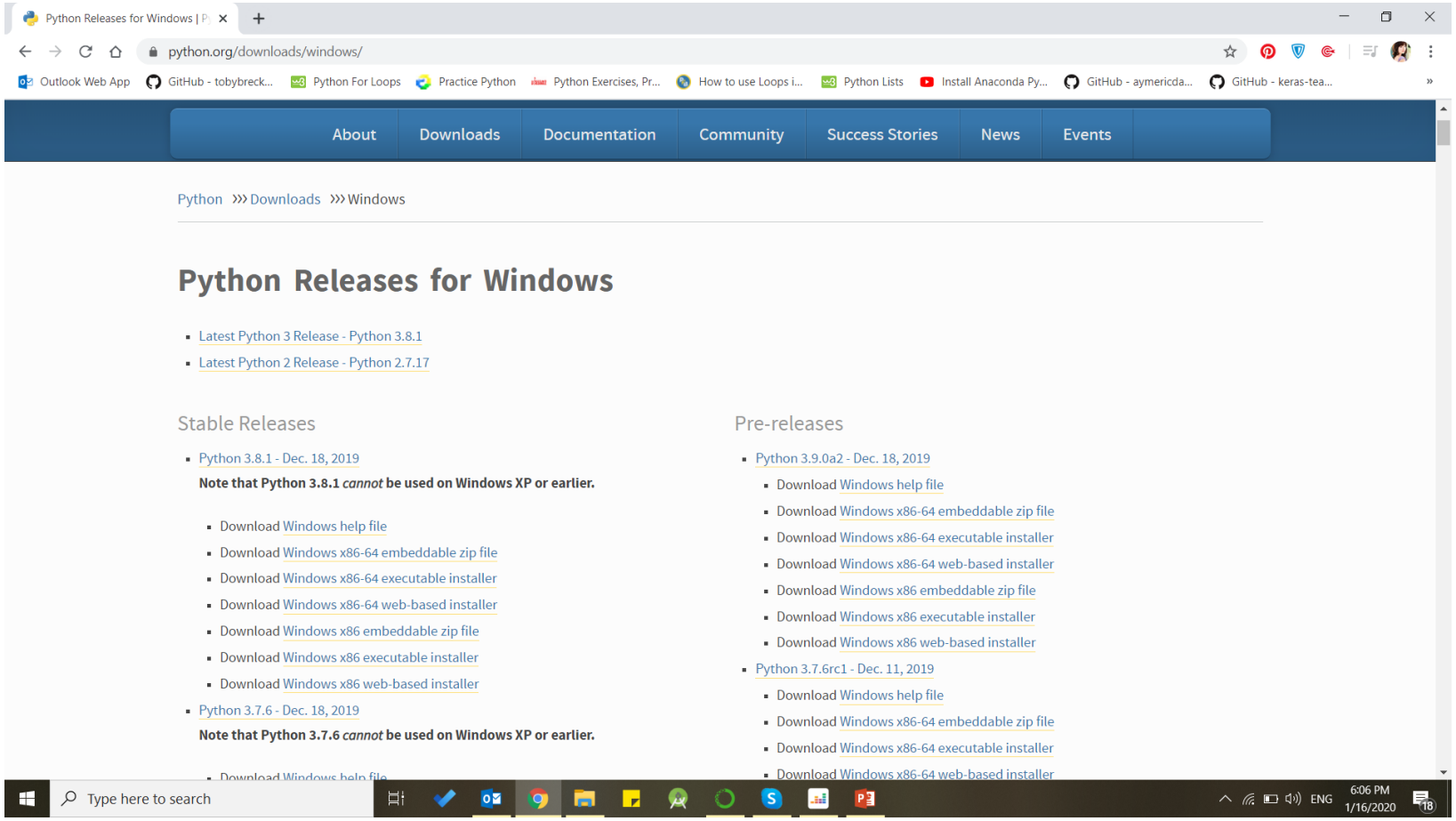
- **Linux**

  $sudo apt-get install python3-minimal

- **Windows**

Download 64 bit python

- Python on default download is 32 bit hence we have to download a specific release and version from python download release section.
- Hence chose your respective python version as per your requirements.



What is PIP

- PIP is a package manager for Python packages, or modules.
- You use PIP to install and update packages and modules as per required by your python code.
- You can check if PIP is Installed by typing following command
  - C:\Users*Your Name*\AppData\Local\Programs\Python\Python36-32\Scripts>pip –version
- If you do not have PIP installed, you can download and install it from,
  - https://pypi.org/project/pip/

Download Package

- Downloading a package in python is very easy.
  - Open the command line interface and tell PIP to download the package you want.
  - Make sure to navigate your command line to the location of Python's script directory, and type the following:
    - pip install packageName

Example

Download a package named "camelcase":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip install camelcase
```

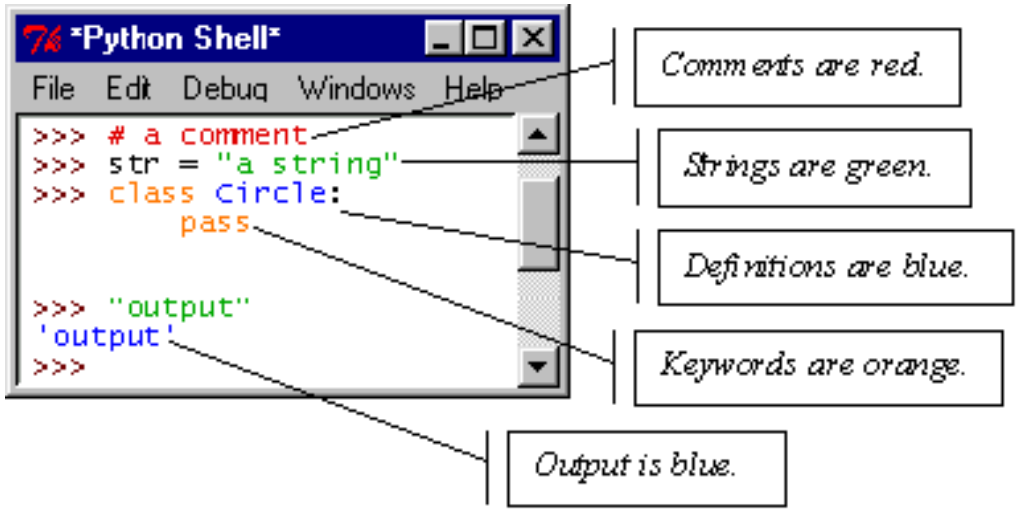
Python Development Interfaces

- **Python Shell** – running 'python' from the Command Line opens this interactive shell
- **IDLE** – a cross-platform Python development environment
- **Spyder-** It is a scientific integrated development environment written in Python, available through Anaconda
- **PyCharm** - a cross-platform IDE, that can be used on Windows, macOS, and Linux.
- **Sublime Text 3** - Sublime Text 3 is a code editor which supports many languages including Python
- **Visual Studio Code** - The editor provides smart code completion based on function definition, imported modules, as well as variable types
- **PythonWin** – a Windows only interface to Python, It provides a simple graphical interface for editing and running Python programs
- **Jupyter-** It supports for Numerical simulation, data cleaning machine learning data visualization, and statistical modelling.

- For the exercises, we'll use Spyder, but you can try them all and pick a favorite

IDLE – Integrated Development Environment

- IDLE helps you program in Python by:
  - color-coding your program code
  - debugging
  - auto-indent
  - interactive shell



Python Programming Modes

1. Interpreter Mode
2. Normal Mode (Script Mode)

Python Programming Modes

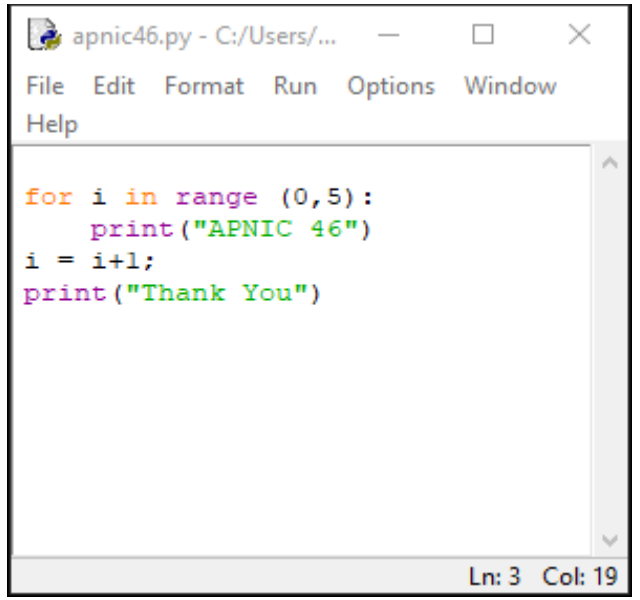
Interactive mode is a command line shell which gives immediate feedback for each statement

The “>>>” indicates that the shell is ready to accept interactive commands

```
root@eve-ng:~# python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>>
>>> 5*9
45
>>> a=5
>>> b=8
>>> a+b
13
>>>
```

Python Programming Modes

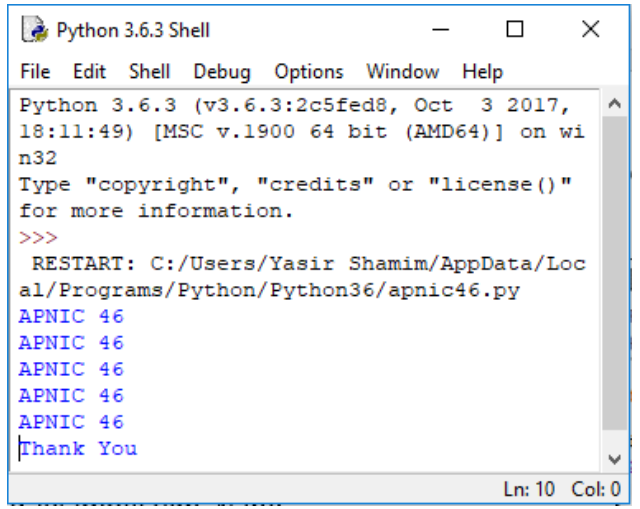
Normal mode is the mode where the scripted (.py) files are run in the Python interpreter.



Python programs are nothing more than text files, and they may be edited with a standard text editor program. For example Vim, Nano, Notepad++,Sublime Text

Instead of having to run one line or block of code at a time, you can type up all your code in one text file, or script, and run all the code at once.

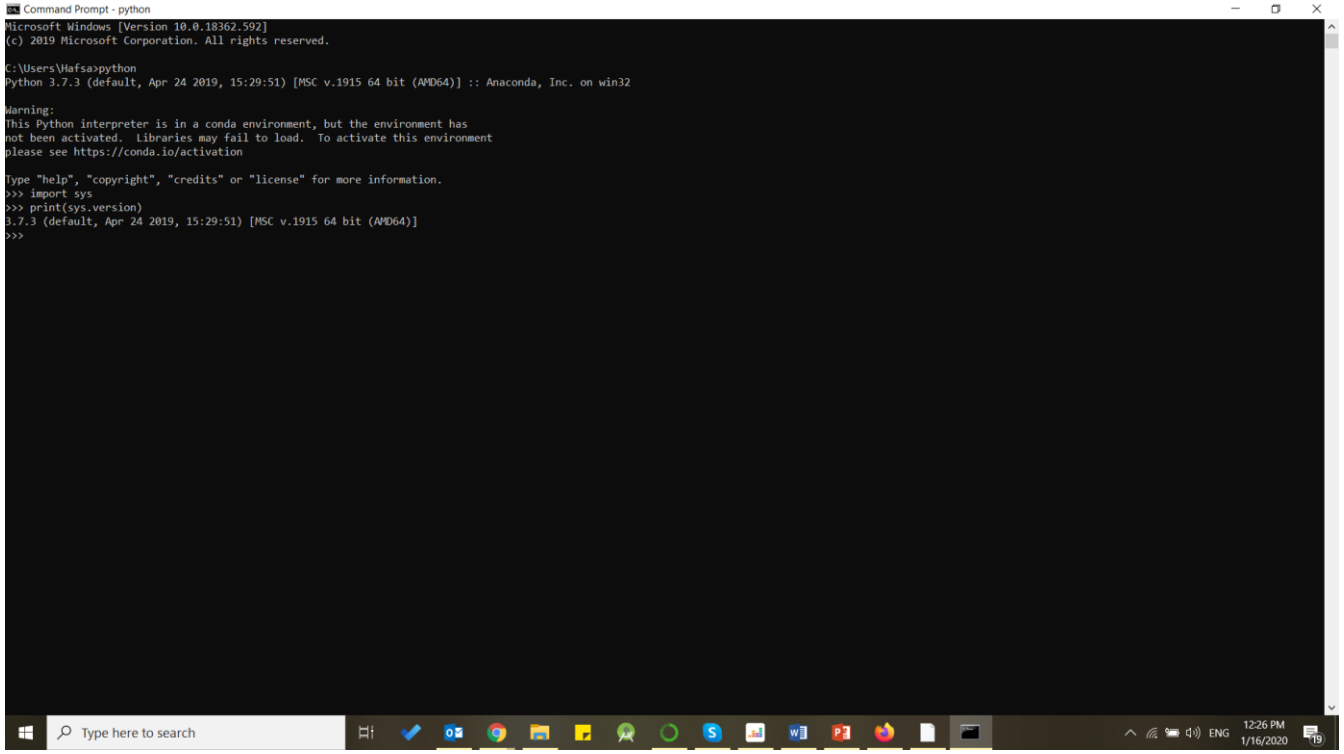
To run the script, either select “Run” -> “Run Module” or press F5.



One more way of execute the python (.py) file from Linux distribution is

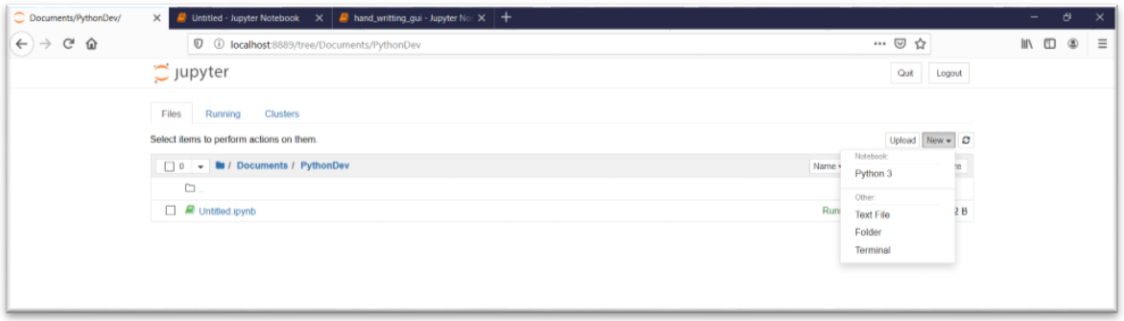
Python Shell

- command line shell is one of the most basic interface for python development.
- You can specifically use this environment for immediate testing of new libraries installed



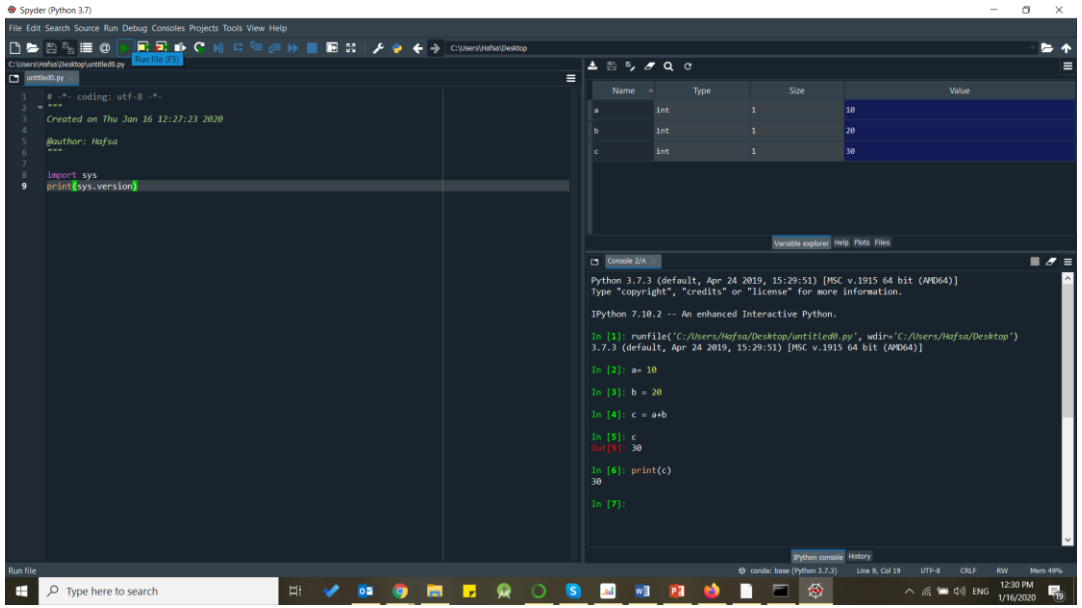
Python Jupyter

- You can add new python jupyter files
- Major advantage of jupyter is that you can execute your code line by line or chunk by chunk
- You don’t need to create a separate file for development and separate shell environment for outputs



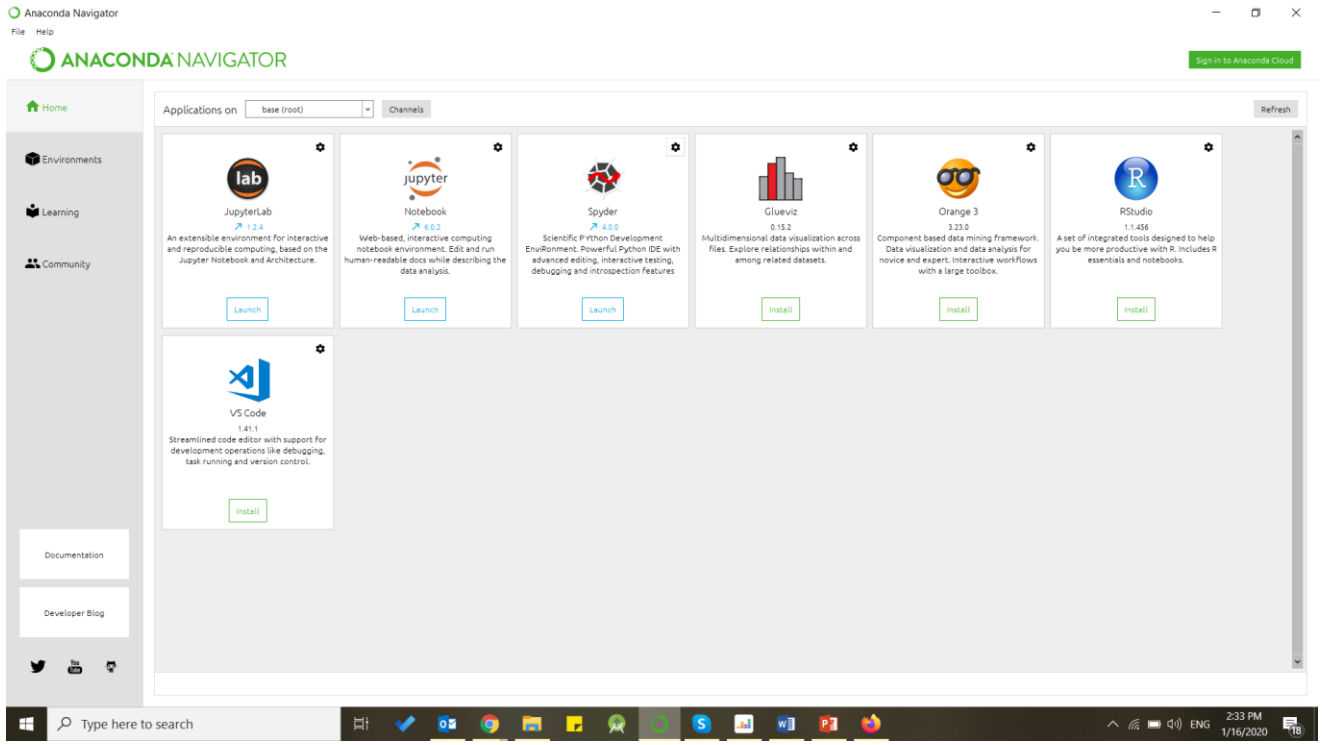
Python Spyder

- Allows you to run Python code by cell, line, or file.
- Plot a histogram or time-series, make changes in date frame.
- It offers automatic code completion and horizontal/vertical splitting.
- Find and eliminate bottlenecks
- An interactive way to trace each step of Python code execution.



Installing an IDE

- Anaconda is a free and open-source distribution of the Python programming language for scientific computing, that aims to simplify package management and deployment
- Anaconda Navigator is a desktop graphical user interface (GUI) included in Anaconda distribution that allows users to launch applications and manage conda packages, environments and channels without using command-line commands.



Online Python IDE

- https://repl.it/

Deeper into Python

Deeper into Python

Variable :

Variables are reserved memory locations to store values.

Numbers | Strings | List | Tuple | Dictionary

Operators :

Operators are the constructs which can manipulate the value of operands.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Deeper into Python

Control Flow Tools :

These are the tools which control the logical flow of the program based on the following parameters.

- Value : Basic units of data. Eg- 10,'string'.
- Variable : A name that refers to a value. Eg- var=10, var is the variable.
- Statement : A section of code that represents a command or action.
- Operator : A symbol that performs operations on operands. Eg- * is for multiplication
- Expression : A combination of variables, operators, and values to perform a task.

Types:

- if statement
- while statement
- for statement
- break, continue and pass

Deeper into Python

Iteration:

Computers are often used to automate repetitive tasks. Repeated execution of a set of statements is called iteration.

Conditionals:

In python conditional statements are features which perform different computations or actions depending on whether a programmer specified boolean condition evaluates to true or false

Regular Expressions:

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern

Deeper into Python

Function:

A function is a block of organized, reusable code that is used to perform a single or multiple **related** action

Classes:
Python uses classes to keep related things together.

Modules:

A Python module is a Python source file which can expose classes, functions and global variables. A Python package is a directory of Python module

Repeatable Code (Writing Reusable Code):

programmers like to be lazy. If something has been done before, why should you do it again?

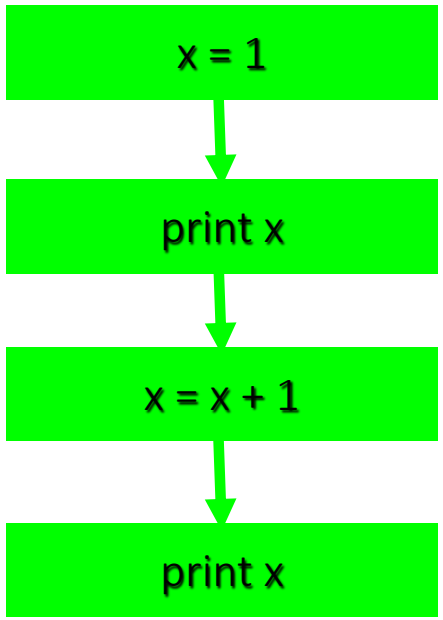
That is what functions cover in python. You've had your code do something and now you want to do it again. You put that code into a function, and re-use it. You can refer to a function anywhere in your code, and the computer will always know what you are talking about.

Python semantics

- Each statement has its own semantics, the def statement doesn't get executed immediately like other statements
- Python uses duck typing, or latent typing
  - Allows for polymorphism without inheritance
  - This means you can just declare
    "somevariable = 67" don't actually have to declare a type
  - print "somevariable = " + tostring(somevariable)"
    strong typing , can't do operations on objects not defined without explicitly asking the operation to be done

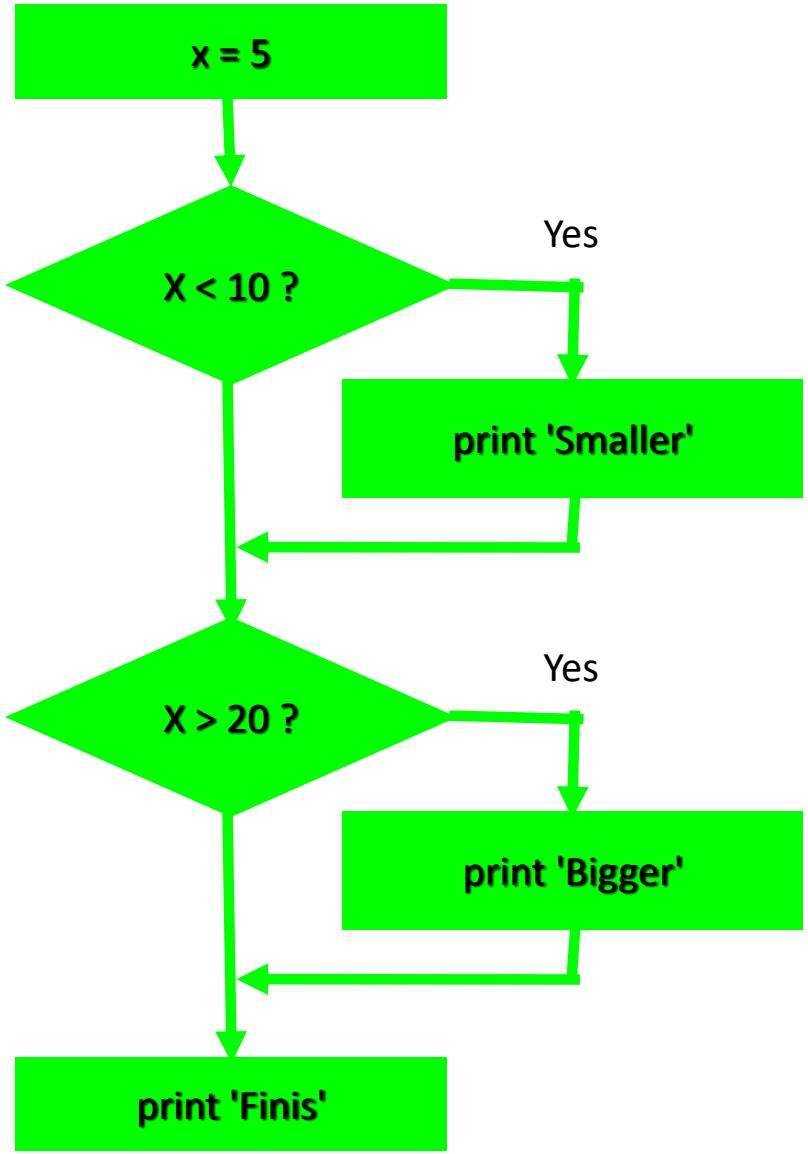
Sequential Steps

When a program is running, it flows from one step to the next. We as programmers set up "paths" for the program to follow.



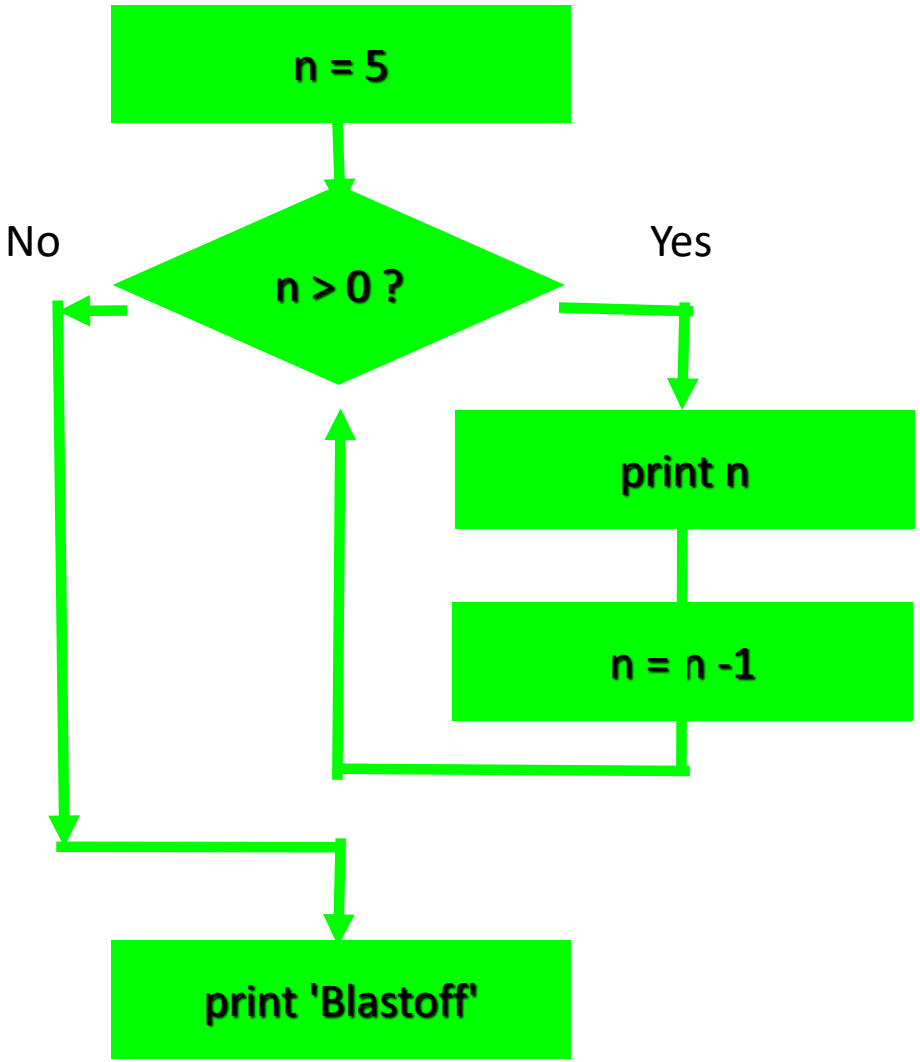

```
x = 1
print x
x = x + 1
print x
```

Conditional Steps



When a program is running, it flows from one step to the next step based on the condition designed.

Repeated Steps



Loops (repeated steps) have iteration variables that change each time through a loop. Often these iteration variables go through a sequence of numbers.

Program:

```
n = 5
while n > 0 :
    print n
    n = n – 1
print 'Blastoff!'
```

Variables Types

- Variables
- Constants

Variables

A variable is a named place in the memory where a programmer can store data and later retrieve the data using the variable “name”

Programmers get to choose the names of the variables

You can change the contents of a variable in a later statement

```
x = 12.2
y = 14
```

Constants

Fixed values such as numbers, letters, and strings are called “constants” - because their value does not change

Numeric constants are as you expect

String constants use single-quotes (')
or double-quotes (")

```
>>> print 123
123
>>> print 98.6
98.6
>>> print 'Hello world'
Hello world
```

Data Types

In Python variables, literals, and constants have a “data type”

In Python variables are “dynamically” typed. In some other languages you have to explicitly declare the type before you use the variable

In C/C++:

```
int a;
float b;
a = 5
b = 0.43
```

In Python:

```
a = 5
a = “Hello”
a = [ 5, 2, 1]
```

Some Data Types in Python

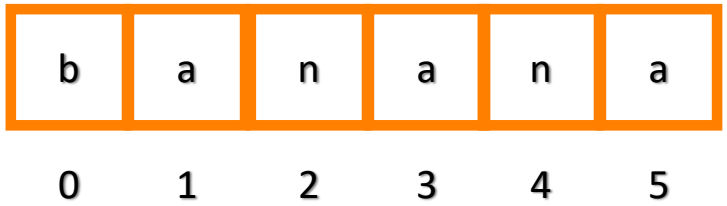
- Integer (Examples: 0, 12, 5, -5)
- Float (Examples: 4.5, 3.99, 0.1)
- String (Examples: “Hi”, “Hello”, “Hi there!”)
- Boolean (Examples: True, False)
- List (Example: [“hi”, “there”, “you”])
- Tuple (Example: (4, 2, 7, 3))

String Data Type

- A string is a sequence of characters
- A string literal uses quotes ‘Hello’ or “Hello”
- For strings, + means “concatenate”
- When a string contains numbers, it is still a string

```
>>> str1 = "Hello“
>>> str2 = 'there‘
>>> bob = str1 + str2
>>> print bob
Hellothere
>>> str3 = '123‘
>>> str3 = str3 + 1
Traceback (most recent call last):  File "<stdin>", line 1, in
<module>TypeError: cannot concatenate 'str' and 'int' objects
```

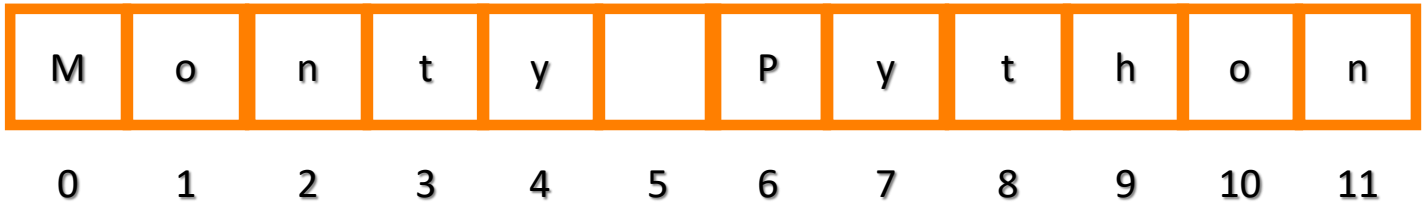
Looking Inside Strings



- We can get at any single character in a string using an index specified in square brackets
- The index value must be an integer and starts at zero
- The index value can be an expression that is computed

```
>>> fruit = 'banana‘
>>> letter = fruit[1]
>>> print letter
a
>>> n = 3
>>> w = fruit[n - 1]
>>> print w
n
```

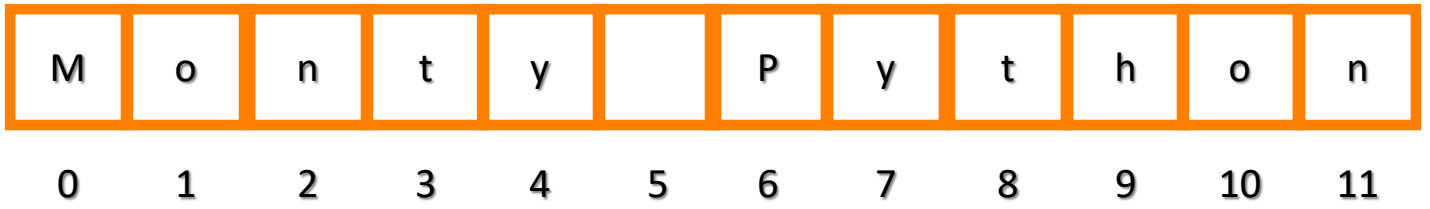
Slicing Strings



- We can also look at any continuous section of a string using a colon operator
- The second number is one beyond the end of the slice - “up to but not including”
- If the second number is beyond the end of the string, it stops at the end

```
>>> s = 'Monty Python‘
>>> print s[0:4]
Mont
>>> print s[6:7]
P
>>> print s[6:20]
Python
```

Slicing Strings



- If we leave off the first number or the last number of the slice, it is assumed to be the beginning or end of the string respectively

```
>>> s = 'Monty Python‘
>>> print s[:2]
Mo
>>> print s[8:]
thon
>>> print s[:]
Monty Python
```

String Concatenation

- When the + operator is applied to strings, it means "concatenation"

```
>>> a = 'Hello‘
>>> b = a + 'There‘
>>> print b
HelloThere

>>> c = a + ' ' + 'There‘
>>> print c
Hello There
```

Handling User Input

- We prefer to read data in using strings and then parse and convert the data as we need
- This gives us more control over error situations and/or bad user input
- Raw input numbers must be converted from strings

```
>>> name = raw_input('Enter:')
Enter:Chuck
>>> print name
Chuck
>>> apple = raw_input('Enter:')
Enter:100
>>> x = apple – 10
Traceback (most recent call last):  File "<stdin>", line 1, in
<module>TypeError: unsupported operand type(s) for -: 'str' and 'int‘
>>> x = int(apple) – 10
>>> print x
90
```

Casting

- Int()
- Float()
- Str()

Integers:

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

Floats:

```
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
```

Strings:

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

Python Collections (Arrays)

- There are four collection data types in the Python programming language:
- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.
- When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Lists

- A list is a collection which is ordered and changeable. In Python lists are written with square brackets.
- You access the list items by referring to the index number.
- You can specify a range of indexes by specifying where to start and where to end the range.
- When specifying a range, the return value will be a new list with the specified items.

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

Tuples

- A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.
- You can access tuple items by referring to the index number, inside square brackets.
- Negative indexing means beginning from the end.
- You can specify a range of indexes by specifying where to start and where to end the range.
- When specifying a range, the return value will be a new tuple with the specified items

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Sets

- A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.
- You cannot access items in a set by referring to an index, since sets are unordered the items has no index.
- But you can loop through the set items
- Once a set is created, you cannot change its items, but you can add new items.

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Dictionary

- A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.
- You can access the items of a dictionary by referring to its key name, inside square brackets.
- You can change the value of a specific item by referring to its key name.
- You can loop through a dictionary.
- When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Operators

- Arithmetic Operators
- Assignment Operators
- Logical Operators
- Membership Operators

Arithmetic Operators

Symbol	Operator Name	Description
+	Addition	Adds the values on either side of the operator and calculate a result.
-	Subtraction	Subtracts values of right side operand from left side operand.
*	Multiplication	Multiplies the values on both sides of the operator.
/	Division	Divides left side operand with right side operand.
%	Modulus	It returns the remainder by dividing the left side operand with right side operand
**	Exponent	Calculates the exponential power

Assignment Operators

Symbol	Operator Name	Description
=	Equal	Assigns the values of right side operand to left side operand.
+=	Add AND	Adds right side operand value to the left side operand value and assigns the results to the left operand.
-=	Subtract AND	Subtracts right side operand value to the left side operand value and assigns the results to the left operand.
*=	Multiply AND	Similarly does their respective operations and assigns the operator value to the left operand.

Logical Operators

Symbol	Operator Name	Description
or	Logical OR	If any of the two operands are non-zero, then the condition is true.
and	Logical AND	If both the operands are true, then the condition is true.
not	Logical NOT	It is used to reverse the logical state of its operand.

Membership Operators

- In
  - The result of this operation becomes True if it finds a value in a specified sequence and False otherwise.
- Not in
  - The result of this operation becomes True if it doesn't find a value in a specified sequence and False otherwise.

Decision making

- if Statement
- if else Statements
- elif Statements

Decision making

Syntax:

```
if expression:
   #execute your code
```

Syntax:

```
if expression:
   #execute your code
else:
   #execute your code
```

Syntax:

```
if expression:
   #execute your code
elif expression:
   #execute your code
else:
   #execute your code
```

Decision making

Example:

```
a = 15
b = 20

if a > b:
   print("a is greater")
else:
   print("b is greater")
```

Output:

```
b is greater
```

Example:

```
a = 15
b = 15

if a > b:
   print("a is greater")
elif a == b:
   print("both are equal")
else:
   print("b is greater")
```

Output:

```
both are equal
```

Example:

```
str1 = input('Please enter first string: ')
str2 = input('Please enter second string: ')
if str2 in str1:
    print(str2+' found in the first string.')
else:
    print(str2+' not found in the first string.')
```

Output:

```
Please enter first string: We are writing core python
Please enter second string: python
python found in the first string.
```

Loops

- For
- While
- Nested for

Looping with For

- We could use a for loop to perform geoprocessing tasks on each layer in a list
- We could get a list of features in a feature class and loop over each, checking attributes
- Anything in a sequence or list can be used in a For loop
- Just be sure not to modify the list while looping

Looping with For

- For allows you to loop over a block of code a set number of times

- For is great for manipulating lists:

```
a = ['cat', 'window', 'defenestrate']
for x in a:
                    print x, len(x)
```

Results:

```
cat 3
window 6
defenestrate 12
```

For loop

Syntax:

```
for iterating_var in sequence:
   #execute your code
```

Example 01:

```
for x in range (0,3) :
    print ('Loop execution %d' % (x))
```

Output:

```
Loop execution 0
Loop execution 1
Loop execution 2
```

While loop

Syntax:

```
while expression:
   #execute your code
```

Example:

```
#initialize count variable to 1
count =1

while count < 6 :
    print (count)
    count+=1
#the above line means count = count + 1
```

Output:

```
1
2
3
4
5
```

Nested Loops

.

Syntax:

```
for iterating_var in sequence:
   for iterating_var in sequence:
      #execute your code
         #execute your code
```

Example:

```
for g in range(1, 6):
    for k in range(1, 3):
        print ("%d * %d = %d" % ( g, k, g*k))
```

Output:

```
1 * 1 = 1
1 * 2 = 2
2 * 1 = 2
2 * 2 = 4
3 * 1 = 3
3 * 2 = 6
4 * 1 = 4
4 * 2 = 8
5 * 1 = 5
5 * 2 = 10
```

Functions in Python

- There are two kinds of functions in Python
- Built-in functions that are provided as part of Python - raw_input(), type(), float(), max(), min(), int(), str(), …
- Functions (user defined) that we define ourselves and then use
- We treat the built-in function names like reserved words (i.e. we avoid them as variable names)

Definition of Function

- In Python a function is some reusable code that takes arguments(s) as input does some computation and then returns a result or results
- We define a function using the def reserved word
- We call/invoke the function by using the function name, parenthesis and arguments in an expression

Functions

- Using 'def' statement for defining a function

Example:

```
def avrg(first, *rests):
    return (first + sum(rests)) / (1 + len(rests))

# Sample use, Putting values

print (avrg(1, 2))
print (avrg(1, 2, 3))
print (avrg(1,2,3,4))
```

Output:

```
1.5
2.0
2.5
```

The range() function

- range() is a built-in function that allows you to create a sequence of numbers in a range
- Very useful in “for” loops which are discussed later in the Iteration chapter
- Takes as an input 1, 2, or 3 arguments. See examples.

```
x = range(5)
print x
[0, 1, 2, 3, 4]
```

```
x = range(3, 7)
print x
[3, 4, 5, 6]
```

```
x = range(10, 1, -2)
print x
[10, 8, 6, 4, 2]
```

Modules

- Modules are functions and variables defined in separate files

- Items are imported using from or import
    - from module import function
    - function()
    - import module
    - module.function()

- Modules are namespaces
    - Can be used to organize variable names, i.e.
        - atom.position = atom.position - molecule.position

Why use Modules?

- **Code reuse**
  - Routines can be called multiple times within a program
  - Routines can be used from multiple programs

- **Namespace partitioning**
  - Group data together with functions used for that data

- **Implementing shared services or data**
  - Can provide global data structure that is accessed by multiple subprograms

OOP Terminology

- **class** -- a template for building objects
- **instance** -- an object created from the template (an instance of the class)
- **method** -- a function that is part of the object and acts on instances directly
- **constructor** -- special "method" that creates new instances

"Special" methods

- All start and end with __ (two underscores)

- Most are used to emulate functionality of built-in types in user-defined classes

- *e.g.* operator overloading
  - **__add__, __sub__, __mult__, ...**
  - see python docs for more information

Classes

- A Class is like an object constructor, or a "blueprint" for creating objects.

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Defining a Class

```
class Thing:

    """This class stores an arbitrary object."""

    def __init__(self, value):
        """Initialize a Thing."""
        self.value = value
    def showme(self):
        """Print this object to stdout."""
        print "value = %s" % self.value
```

constructor

method

Using a Class

- **t = Thing(10) # calls __init__ method**
- **t.showme() # prints "value = 10"**
- **t** is an instance of class **Thing**
- **showme** is a method of class **Thing**
- **__init__** is the constructor method of class **Thing**
- when a **Thing** is created, the **__init__** method is called
- Methods starting and ending with __ are "special" methods

Using a Class

print t.value # prints "10"

value is a *field* of class **Thing**

t.value = 20   # change the field value

print t.value # prints "20"

File Handling

- Files are manipulated by creating a file object
  - f = open("points.txt", "r")

- The file object then has new methods
  - print f.readline() *# prints line from file*

- Files can be accessed to read or write
  - f = open("output.txt", "w")
  - f.write("Important Output!")

- Files are iterable objects, like lists

File Opening and Attributes

Syntax:

```
file_object = open(filename [,mode] [,buffering])
```

Example:

```
# file opening example in Python
fo = open("sample.txt", "wb")
    print ("File Name: ", fo.name)
    print ("Mode of Opening: ", fo.mode)
    print ("Is Closed: ", fo.closed)
    print ("Softspace flag : ", fo.softspace)
```

Output:

```
File Name: sample.txt
Mode of Opening: wb
Is Closed: False
Softspace flag: 0
```

Modes of File opening

r	Opens a file for reading only. (It's a default mode.)
w	Opens a file for writing. (If a file doesn't exist already, then it creates a new file. Otherwise, it's truncate a file.)
x	Opens a file for exclusive creation. (Operation fails if a file does not exist in the location.)
a	Opens a file for appending at the end of the file without truncating it. (Creates a new file if it does not exist in the location.)
t	Opens a file in text mode. (It's a default mode.)
b	Opens a file in binary mode.
+	Opens a file for updating (reading and writing.)

File Reading / Writing

Example:

```
# read the entire file as one string
with open('filename.txt') as f:
data = f.read()

# Iterate over the lines of the File
with open('filename.txt') as f:
for line in f :
   print(line, end=' ')
# process the lines
```

Example:

```
# Write text data to a file
with open('filename.txt' , 'wt') as f:
    f.write ('hi there, this is a first line of file.\n')
    f.write ('and another line.\n')
```

Output:

```
hi there, this is a first line of file.
and another line.
```

Time Library

- Python has a module named “time” to handle time-related tasks. To use functions defined in the module, we need to import the module first.
- Time function returns the number of seconds passed since epoch
- Ctime function takes seconds passed since epoch as an argument and returns a string representing local time
- Sleep function suspends (delays) execution of the current thread for the given number of seconds.

```
import time
seconds = time.time()
print("Seconds since epoch =", seconds)
```

Types of errors

- **IndexError** is thrown when trying to access an item at an invalid index.
- **ModuleNotFoundError** is thrown when a module could not be found.
- **KeyError** is thrown when a key is not found.
- **ImportError** is thrown when a specified function can not be found.
- **TypeError** is thrown when an operation or function is applied to an object of an inappropriate type.
- **ValueError** is thrown when a function's argument is of an inappropriate type.
- **NameError** is thrown when an object could not be found.
- **ZeroDivisionError** is thrown when the second operator in the division is zero.
- **KeyboardInterrupt** is thrown when the user hits the interrupt key (normally Control-C) during the execution of the program.

Exception handling

- try/except
  - catch the error and recover from exceptions hoist by programmers or Python itself.
- try/finally
  - Whether exception occurs or not, it automatically performs the clean-up action.
- Assert
  - triggers an exception conditionally in the code.

Example 01:

```
(a,b) = (6,0)
try:# simple use of try-except block for handling errors
    g = a/b
except ZeroDivisionError:
    print ("This is a DIVIDED BY ZERO error")
```

Output:

```
This is a DIVIDED BY ZERO error
```

Example 02:

```
(a,b) = (6,0)
try:
    g = a/b
except ZeroDivisionError as s:
    k = s
    print (k)
#Output will be: integer division or modulo by zero
```

Output:

```
division by zero
```

Comments

- Single-Line Comment

Example:

```
#Defining a variable to store number.
n = 50 #Store 50 as value into variable n.
```

- Multi-Line Comment

Example:

```
"""
Author: www.w3schools.in
Description:
Writes the words Hello World on the screen
"""
```

Indentation Rules

- Increase indent after an if statement or for statement (after :)
- Maintain indent to indicate the scope of the block (which lines are affected by the if/for)
- Reduce indent to back to the level of the if statement or for statement to indicate the end of the block
- Blank lines are ignored - they do not affect indentation
- Comments on a line by themselves are ignored w.r.t. indentation

Indentation and Blocks

- Python uses whitespace and indents to denote blocks of code
- Lines of code that begin a block end in a colon:
- Lines within the code block are indented at the same level
- To end a code block, remove the indentation
- You'll want blocks of code that run only when certain conditions are met

Indentation

Example:

```
if a==1:
    print(a)
    if b==2:
        print(b)
print('end')
```

Example:

```
    print(a)
    if b==2:
```

Whitespace

- Whitespace is meaningful in Python: especially indentation and placement of newlines.

- Use a newline to end a line of code
  Use \ when must go to next line prematurely.

- No braces {} to mark blocks of code, use *consistent* indentation instead
  - First line with *less* indentation is outside of the block.
  - First line with *more* indentation starts a nested block.

- Colons start of a new block in many constructs, e.g. function definitions, then clauses

Naming Rules

- Names are case sensitive and cannot start with a number.

- They can contain letters, numbers, and underscores.

  ```
  bob  Bob  _bob  _2_bob_  bob_2  BoB
  ```

- There are some reserved words:

  and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while

Naming conventions

- The Python community has these recommend-ed naming conventions.
- joined_lower for functions, methods and, attributes.
- joined_lower or ALL_CAPS for constants.
- StudlyCaps for classes.
- camelCase only to conform to pre-existing conventions.
- Attributes: interface, _internal, __private

Accessing Non-Existent Name

Accessing a name before it's been properly created (by placing it on the left side of an assignment), raises an error

```
>>> y

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    y
NameError: name 'y' is not defined
>>> y = 3
>>> y
3
```

Keywords

and	assert	in
del	else	raise
from	if	continue
not	pass	finally
while	yield	is
as	break	return
elif	except	def
global	import	for
or	print	lambda
with	class	try
exec		

String Methods

Method	Description
lstrip()	Returns a left trim version of the string
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Method	Description
capitalize()	Converts the first character to upper case
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
lower()	Converts a string into lower case

List Methods

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Sets Methods

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

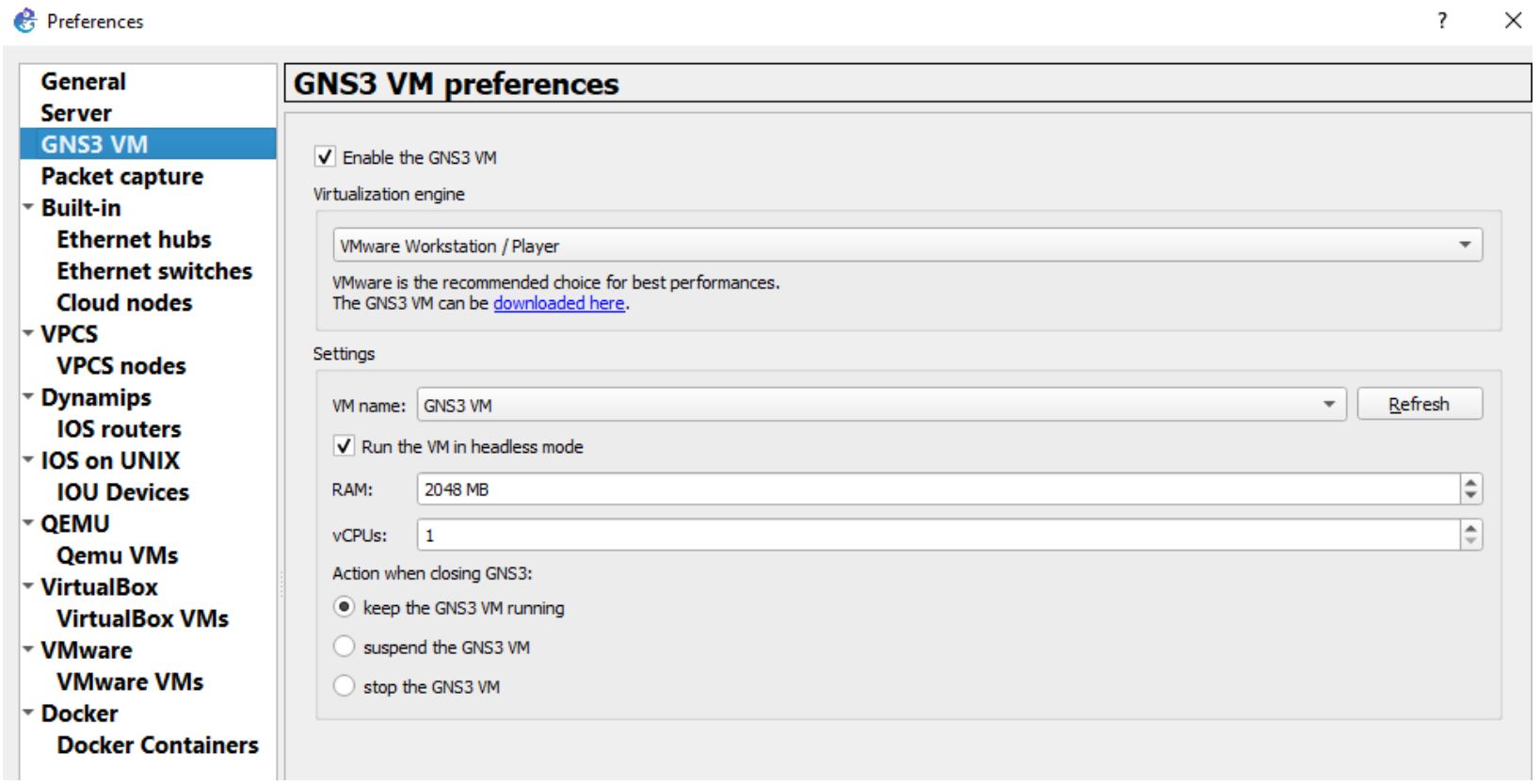
Network Integration with Python

- Telnet
- Netmiko
- NAPALM

GNS3 DEMO

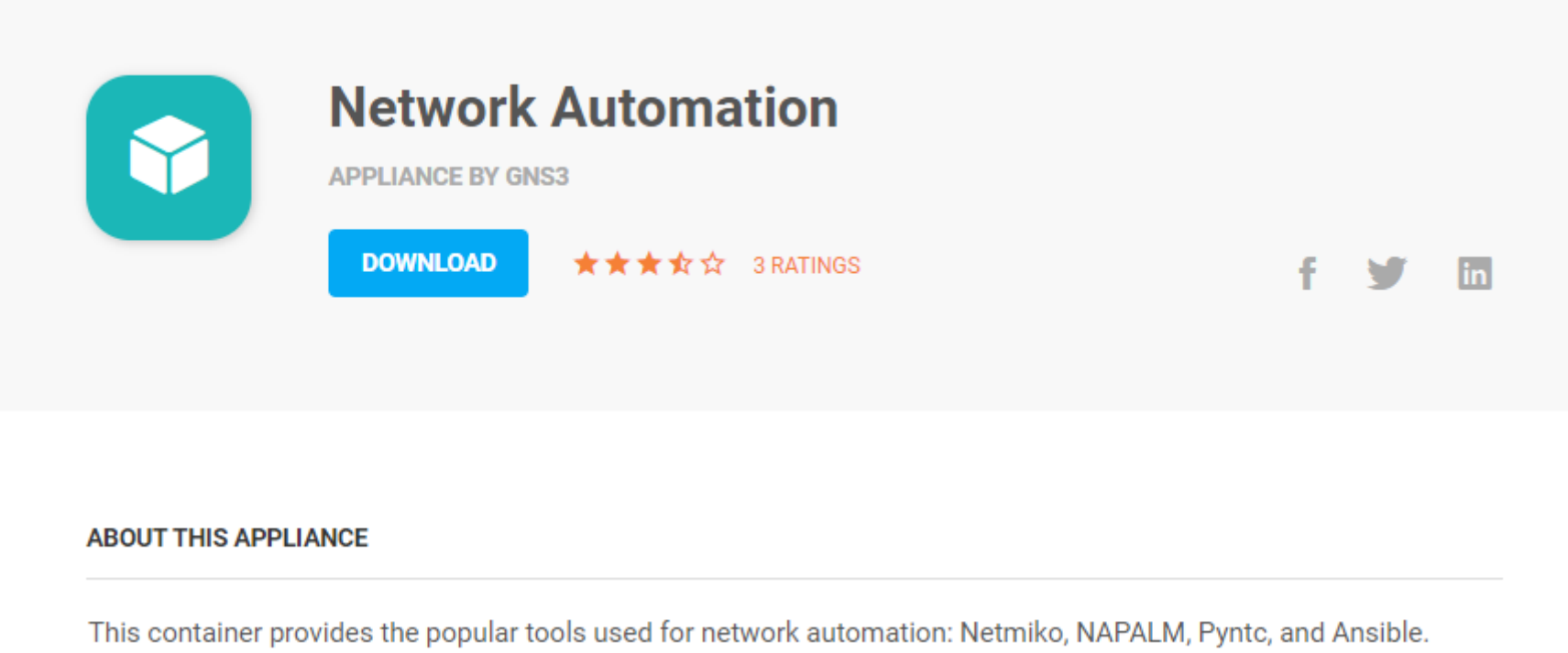
GNS3 DEMO

- Download gns3
  - https://www.gns3.com/
- Download the GNS3 VM
  - Edit/Preference/GNS3VM/download here



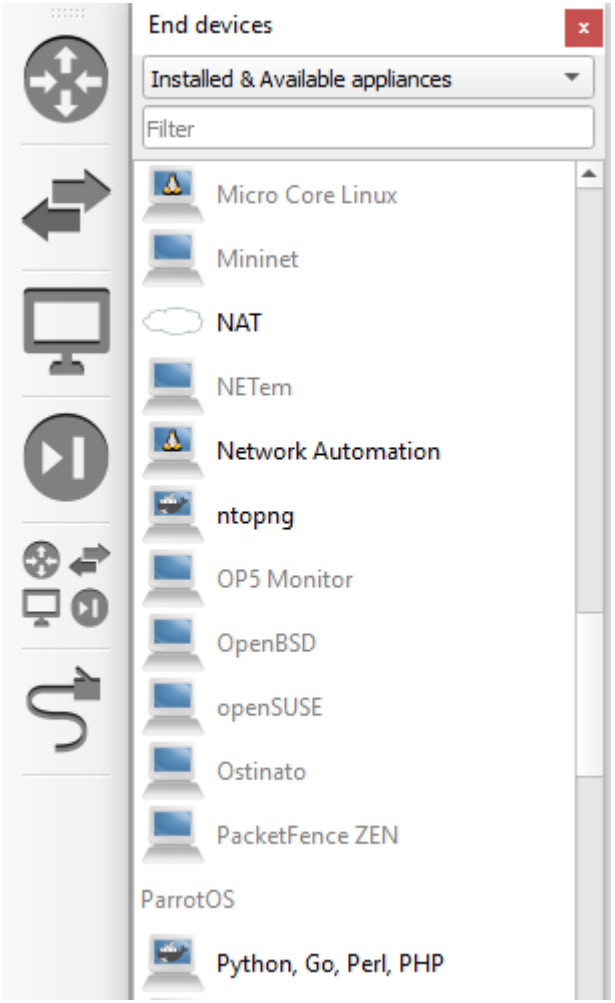
GNS3 DEMO

- Use VMware workstation to import the GNS3VM
- Download the GNS3 appliance
  - https://www.gns3.com/marketplace/appliance/network-automation



GNS3 DEMO

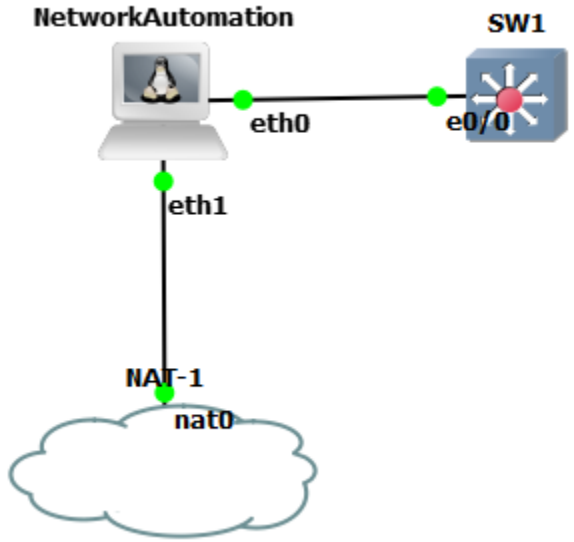
- Import the appliance
  - File/import appliance



GNS3 DEMO

- Drag the Network Automation appliance, NAT Cloud and Switch.

- Modify Network Automation appliance to add one more Network Adapter



Telnet

Telnet

In **Python telnet** is implemented by the module telnetlib which has the **Telnet** class which has the required methods to establish the connection

```
import telnetlib
import time

password = ("cisco")

tn = telnetlib.Telnet("192.168.1.10")
tn.read_until("Password: ")
tn.write(password + "\n")

tn.write("enable \n")
tn.read_until("Password: ")
tn.write(password + "\n")

tn.write("conf t \n")
time.sleep(1)
tn.write("interface loopback10 \n")
time.sleep(1)
tn.write("ip address 10.1.1.1 255.255.255.0 \n")
time.sleep(1)
tn.write("end \n")
time.sleep(1)
tn.write("exit \n")

print tn.read_very_eager()
print("\nThank You")
```

```
root@NetworkAutomation:~# python telnet.py

SW1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW1(config)#interface loopback100
SW1(config-if)#ip address 100.1.1.1 255.255.255.0
SW1(config-if)#end
SW1#
```

Telnet

```
import telnetlib
import time

username = ("cisco")
password = ("cisco")

tn = telnetlib.Telnet("192.168.1.10")
tn.read_until("Username: ")
tn.write(username + "\n")
tn.read_until("Password: ")
tn.write(password + "\n")
tn.write("conf t \n")
time.sleep(1)

for x in range (2,10):
   tn.write("vlan " + str(x) + "\n")
   time.sleep(1)
   tn.write("name vlan_" + str(x) + "\n")
   time.sleep(1)

tn.write("end\n")
time.sleep(1)
tn.write("exit\n")

print tn.read_very_eager()
print("\nThank You")
```

```
root@NetworkAutomation:~# python telnet_loop.py

SW1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW1(config)#vlan 2
SW1(config-vlan)#name vlan_2
SW1(config-vlan)#vlan 3
SW1(config-vlan)#name vlan_3
SW1(config-vlan)#vlan 4
SW1(config-vlan)#name vlan_4
SW1(config-vlan)#vlan 5
SW1(config-vlan)#name vlan_5
SW1(config-vlan)#vlan 6
SW1(config-vlan)#name vlan_6
SW1(config-vlan)#vlan 7
SW1(config-vlan)#name vlan_7
SW1(config-vlan)#vlan 8
SW1(config-vlan)#name vlan_8
SW1(config-vlan)#vlan 9
SW1(config-vlan)#name vlan_9
SW1(config-vlan)#end
SW1#

Thank You
```

Netmiko

Netmiko

The purposes of this library are the following:

- Successfully establish an SSH connection to the device
- Simplify the execution of show commands and the retrieval of output data
- Simplify execution of configuration commands including possibly commit actions

Do the above across a broad set of networking vendors and platforms

Netmiko

```
from netmiko import ConnectHandler

ios = {
    'device_type': 'cisco_ios',
    'ip': '192.168.1.10',
    'username': 'cisco',
    'password': 'cisco'
}

net_connect = ConnectHandler(**ios)
output = net_connect.send_command('show ip int brief')
print (output)

config_commands = ['int loop 0', 'ip address 1.1.1.1 255.255.255.0']
output = net_connect.send_config_set(config_commands)
print (output)
```

```
root@NetworkAutomation:~# python netmiko1.py
Interface              IP-Address      OK? Method Status                Protocol
Ethernet0/0            unassigned      YES unset  up                    up
Ethernet0/1            unassigned      YES unset  up                    up
Ethernet0/2            unassigned      YES unset  up                    up
Ethernet0/3            unassigned      YES unset  up                    up
Ethernet1/0            unassigned      YES unset  up                    up
Ethernet1/1            unassigned      YES unset  up                    up
Ethernet1/2            unassigned      YES unset  up                    up
Ethernet1/3            unassigned      YES unset  up                    up
Ethernet2/0            unassigned      YES unset  up                    up
Ethernet2/1            unassigned      YES unset  up                    up
Ethernet2/2            unassigned      YES unset  up                    up
Ethernet2/3            unassigned      YES unset  up                    up
Ethernet3/0            unassigned      YES unset  up                    up
Ethernet3/1            unassigned      YES unset  up                    up
Ethernet3/2            unassigned      YES unset  up                    up
Ethernet3/3            unassigned      YES unset  up                    up
Loopback0              1.1.1.1         YES NVRAM  up                    up
Vlan1                  unassigned      YES unset  administratively down down
Vlan10                 192.168.1.10    YES NVRAM  up                    up
config term
Enter configuration commands, one per line.  End with CNTL/Z.
IOU1(config)#int loop 0
IOU1(config-if)#ip address 1.1.1.1 255.255.255.0
IOU1(config-if)#end
```

NAPALM

NAPALM

NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) is a Python library that implements a set of functions to interact with different network device Operating Systems using a unified API.

NAPALM supports several methods to connect to the devices, to manipulate configurations or to retrieve data.

Supported Network Operating Systems:

- Arista EOS
- Cisco IOS
- Cisco IOS-XR
- Cisco NX-OS
- Juniper JunOS

https://napalm.readthedocs.io/en/latest/index.html

NAPALM

	EOS	IOS	IOSXR	JUNOS	NXOS	NXOS_SSH
get_arp_table	✅	✅	❌	❌	❌	✅
get_bgp_config	✅	✅	✅	✅	❌	❌
get_bgp_neighbors	✅	✅	✅	✅	✅	✅
get_bgp_neighbors_detail	✅	✅	✅	✅	❌	❌
get_config	✅	✅	✅	✅	✅	✅
get_environment	✅	✅	✅	✅	✅	✅
get_facts	✅	✅	✅	✅	✅	✅
get_firewall_policies	❌	❌	❌	❌	❌	❌
get_interfaces	✅	✅	✅	✅	✅	✅
get_interfaces_counters	✅	✅	✅	✅	❌	❌
get_interfaces_ip	✅	✅	✅	✅	✅	✅
get_ipv6_neighbors_table	❌	✅	❌	✅	❌	❌
get_lldp_neighbors	✅	✅	✅	✅	✅	✅
get_lldp_neighbors_detail	✅	✅	✅	✅	✅	✅
get_mac_address_table	✅	✅	✅	✅	✅	✅
get_network_instances	✅	✅	❌	✅	✅	❌

NAPALM

get_ntp_peers	✗	✓	✓	✓	✓	✓
get_ntp_servers	✓	✓	✓	✓	✓	✓
get_ntp_stats	✓	✓	✓	✓	✓	✗
get_optics	✓	✓	✗	✓	✗	✗
get_probes_config	✗	✓	✓	✓	✗	✗
get_probes_results	✗	✗	✓	✓	✗	✗
get_route_to	✓	✓	✓	✓	✗	✓
get_snmp_information	✓	✓	✓	✓	✓	✓
get_users	✓	✓	✓	✓	✓	✓
is_alive	✓	✓	✓	✓	✓	✓
ping	✓	✓	✗	✓	✓	✓
traceroute	✓	✓	✓	✓	✓	✓

NAPALM

```
from napalm import get_network_driver

driver = get_network_driver('ios')

ios = driver('192.168.1.10', 'cisco', 'cisco')

ios.open()

ios_output = ios.get_facts()

print (ios_output)
```

python napalm1.py

{u'os_version': u'Solaris Software (I86BI_LINUXL2-ADVENTERPRISEK9-M),
Experimental Version 15.1(20130726:213425) [dstivers-j
uly26-2013-team_track 104]',
u'uptime': 2640,
u'interface_list': [u'Ethernet0/0', u'Ethernet0/1', u'Ethernet0/2', u'Ethernet
0/3', u'Ethernet1/0', u'Ethernet1/1', u'Ethernet1/2', u'Ethernet1/3',
u'Ethernet2/0', u'Ethernet2/1', u'Ethernet2/2', u'Ethe
rnet2/3', u'Ethernet3/0', u'Ethernet3/1', u'Ethernet3/2', u'Ethernet3/3',
u'Loopback0',
u'Vlan1', u'Vlan10'],
u'vendor': u'Cisco',
u'serial_number': u'2048001', u'model': u'Unknown', u'hostname': u'IOU1',
u'fqdn': u'IOU1.cisco.com'}

Use Case

Case I :
Configure Vlan on multiple switches and interfaces simultaneously.

Thinking Process !

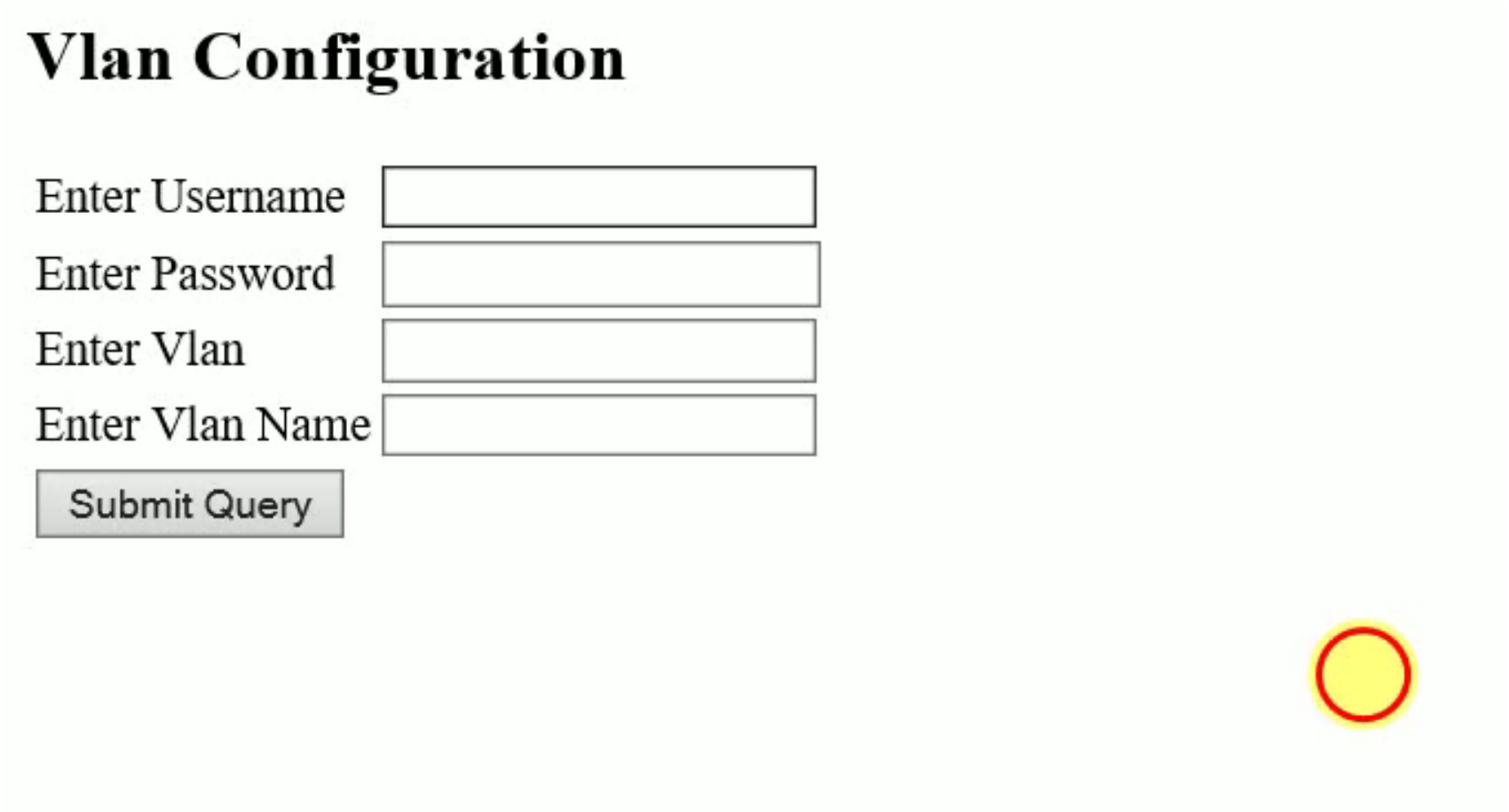
- Problem identification
  - Configure Vlan on multiple switches and its interfaces simultaneously.
  - Error handling.
  - Time mismanagement.
- Implementing a Plan
  - Automate vlan creation.
  - Use programming language
  - Debugging issues
  - Determine format of output
  - Integration of frontend and backend code.
- Solution
  - No more manual configuration
  - Error less provisioning
  - Efficient utilization of time
- Future Enhancements
  - Multi vendor support

Frontend HTML Page

```
<html>

<body>
        <h2>Vlan Configuration</h2>

<form method="post" action="submit.php">
<table>
        <tr>
                <td>
                        <p>Enter Username</p>
                </td>
                <td>
                        <input type="text" name="user">
                </td>
        </tr>

        <tr>
                <td>
                        <p>Enter Password</p>
                </td>
                <td>
                        <input type="password" name="password">
                </td>
        </tr>

        <tr>
                <td>
                        <p>Enter Vlan</p>
                </td>
                <td>
                        <input type="text" name="vlan" maxlength="4" >
                </td>
        </tr>

        <tr>
                <td>
                        <p>Enter Vlan Name</p>
                </td>
                <td>
                        <input type="name" name="name">
                </td>
        </tr>

           <tr>
                <td>

                        <input type=submit> <br>
                </td>


     </tr>
</table>

</body>
</html>
```

```
<!DOCTYPE HTML>

<html>

<body>

<?php

$vlan=$_POST['vlan'];

$name=$_POST['name'];

$user=$_POST['user'];

$password=$_POST['password'];

echo shell_exec("python /var/www/html/vlan/huaweiring.py '$vlan' '$name' '$user' '$password'");

echo nl2br(file_get_contents("/var/www/html/vlan/final"));

file_put_contents('/var/www/html/vlan/final', '');

?>

</body>

</html>
```

Frontend HTML Page

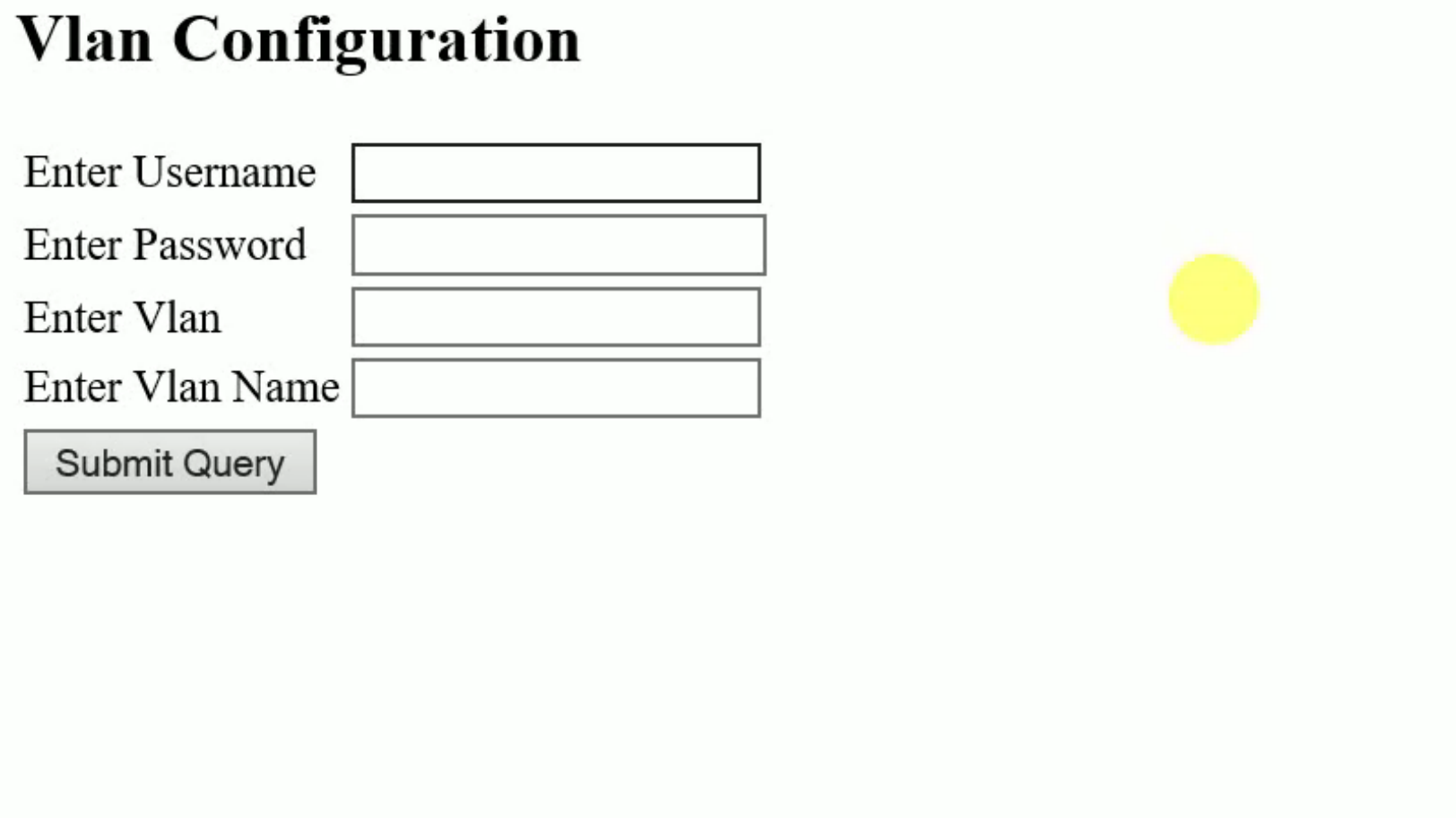


Backend Python Code

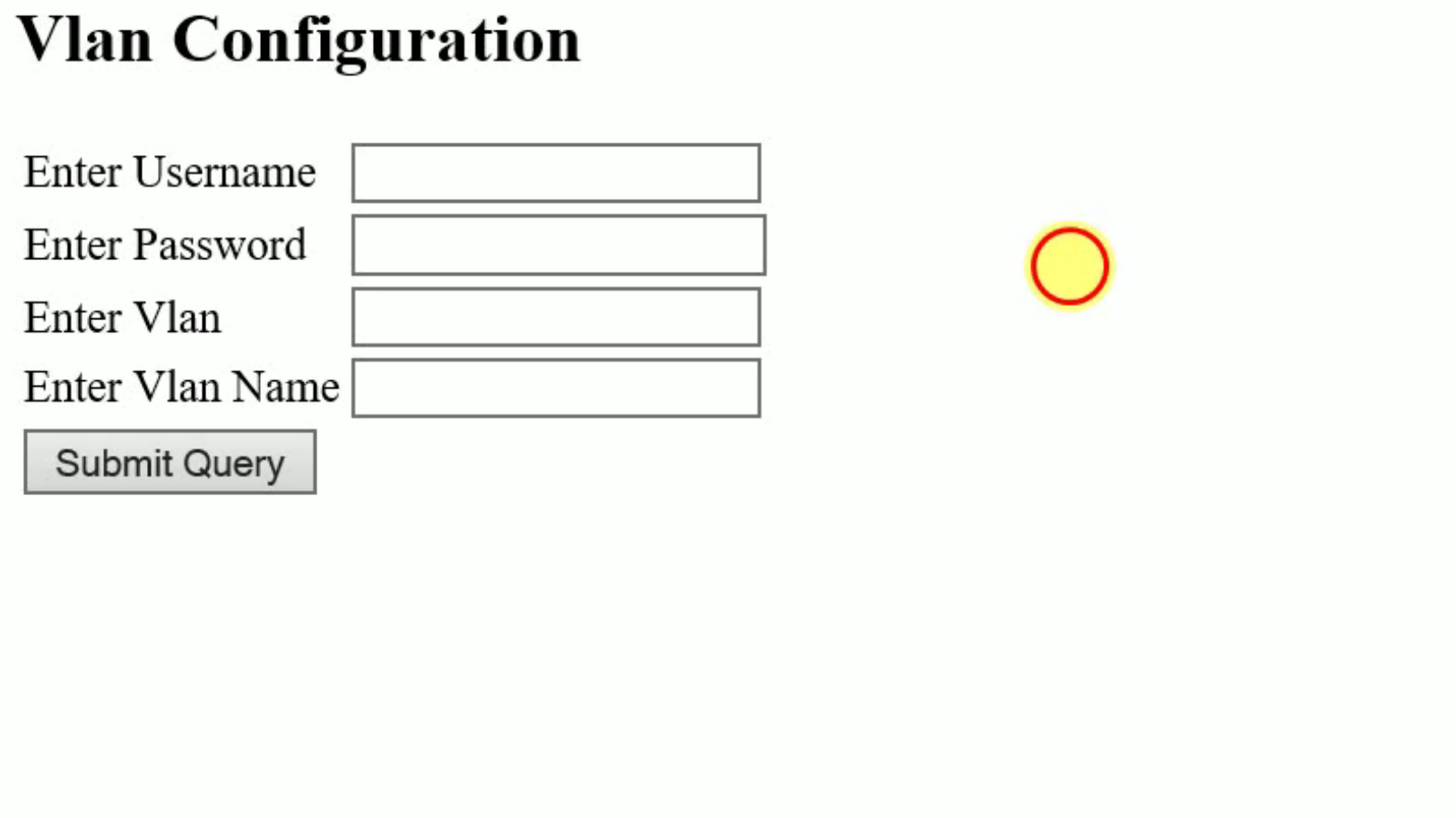
```python
import getpass
import sys
import telnetlib
import time
import re


vlan=sys.argv[1]
name=sys.argv[2]
user=sys.argv[3]
password=sys.argv[4]


vlan0 = int(vlan)

if vlan0 > 1 and vlan0 < 4095:

    mylist=["192.168.93.98"]

    count1=0

    for m in range(0,1):
        tn = telnetlib.Telnet(mylist[count1])

        time.sleep(1)
        tn.write(user + "\n")
        time.sleep(1)
        tn.write(password + "\n")
        time.sleep(1)
        tn.write("screen-length 0 temporary\n")
        tn.write("display vlan\n")

        time.sleep(5)
        output =  tn.read_very_eager()
        tn.write("quit\n")

        with open('output1', 'w') as f:
            for line in output:
                f.write(line)

        vlan1 = open('/var/www/html/vlan/output1', 'r')

        for line1 in vlan1:
            if re.match(vlan, line1):
                break

        with open('/var/www/html/vlan/output2', 'w') as f:
            f.write(line1)

        req = open('/var/www/html/vlan/output2', 'r')

        for vlan2 in req:
            fields = vlan2.strip().split()
            fields1 = fields[0]
            break

        if vlan == fields1:
            print( "<br>vlan id " + str(vlan) + " already exists in " + str(mylist[count1]))
            quit()

        count1 = count1+1

    print("<br><br>You are good to go\n\n")

    count=0
    for n in range(0,1):

        print("<br>")
        print("<br> Configuring Host " + mylist[count] + ("\n"))
        tn = telnetlib.Telnet(mylist[n])
        time.sleep(1)
        tn.write(user + "\n")
        time.sleep(1)
        tn.write(password + "\n")
        time.sleep(1)
        print("vlan about to create")
        tn.write("system-view\n\n\n")
        tn.write("vlan " + str(vlan) + "\n\n")
        print("\n")
        tn.write("description " + str(name) + "\n\n")
        print("\n")
        tn.write("quit\n")
        print("\n")
        tn.write("interface GigabitEthernet0/0/22\n\n")
        print("\n")
        tn.write("port link-type access\n\n")
        print("\n")
        tn.write("port default vlan " + str(vlan) + "\n\n")
        print("\n")
        tn.write("quit\n")
        tn.write("quit\n")
        time.sleep(5)
        final1 = tn.read_very_eager()
        with open('/var/www/html/vlan/final', 'w') as f:
            for line5 in final1:
                f.write(line5)

        count=count+1

    print("<br><br> Thank You !")
    print("<br><br> Verfication Commands:")
else:
    print("<br>vlan out of range")
```

Debug session on Network Node

What happened when vlan already exists !



Enter incorrect vlan !



Python for Network Engineers

- The language which is widely opt by Network community and there is a very big gap between Python and other prevalent languages such as Perl, Ruby, Go as far as the size of the community.
- For new comers make simple scripts then move to complicated ones.
- There is a vast amount of available network libraries.
- Use various Python tools/modules to handle network devices from several networking vendors: Cisco (IOS, IOS XE, IOS XR), Juniper, Arista, HP, Avaya.
- Execute CLI commands on several network devices simultaneously.

Closing Remarks

- If you are really interested in moving forward, try making your mind and picking up a choice.
- Community is there to support you.
- Start small, celebrate the little things, and build, build, build!
- At the end of the day the true reason for happiness is creating something out of nothing.

A good mentor of mine told me if you want to create a flying car, then start with making some wheels into a skateboard, enjoy the skateboard and turn that into a bike and so on. – Yad Faeq

Questions !

Thank You

You will never be 100% ready to change. Don’t wait for the perfect time. It will never come.

Start today !