Getting Started with Python

Python is a general-purpose, high-level programming language. It was created by Guido van Rossum in the late 1980's with its first implementation started in December 1989. Named after the Monty Python's Flying Circus as Guido was a huge fan of the show. Python introduced the idea where indentation was used rather than brackets to specify closures in the code. As Python became popular this indentation became a huge point of contention for new developers migrating from other languages such as PERL. However its benefit is that the code is similarly formatted across multiple developers applications. This was a huge departure from PERL where often there were little coding standards and it made PERL typically hard to maintain.

Python has become the "gold standard" for programming languages. It has a great deal of available libraries that provide support for just about every possible thing that you can imagine. Advanced scientific computing NumPy, gaming pygame, to automation with Ansible. The ability to write code a single time and not having to worry about porting to multiple platforms is a huge advantage. With the ubiquitous nature of Python and its adherence to formatting standards as defined in PEP8 it gives programmers an advantage to solve various domain problems in a simmilar fashon.

It is a dynamically typed language in which you do not have to specify the type of a variable. When Python is run it takes the script file and interprets it into byte code. This can be seen when you run a **.py** file as a **.pyc** file is generated in the same folder. This byte code is then run in a virtual machine, similar to Java. This gives the language the ability to be written once and run across multiple platforms. While this has some performance implications generally Python performs quite well. The language is used heavily in many different fields from math, science simulations, various application services, and of course automation.

Some of the downsides to Python stem from its need to move forward. Python created a new version in the 3.X branch in December of 2008. The goal was to remove some of the languages rough edges and increase overall performance. Adoption for the new version has been slow due to the fact that the code, while similar, is not 100% backwards compatible. Due to the wide variety of packages available in 2.X most developers tend to stick with the older release. In the end the competition between the two versions is positive as it helps drive new features into the 3.X line such as the inclusion of strong-typing (predefined data structures).

Python Language Syntax

Programming comes down to two basic concepts: data structures and algorithms. No matter which language you choose to use these are the fundamentals that will be used. A data structure is the format in which you store data. Each language generally has a similar set of data structures that are used such as dictionaries, a key/value type of structure. This is because no matter what language you use you are looking to solve similar problems. Some languages, such as R (http://www.r-project.org/), have structures that are more friendly to solving its specific problems. In the case of R it has advanced matrices that are used for multidimensional data. The goal of a language is to give the programmer flexibility and yet help them solve problems in an easy manner.

Once you have your data or data structures defined then we need to go about using them. Data is read, manipulated, removed, and created throughout a program. An algorithm is defined as a formula or procedure for solving a problem. Or in simpler terms how we interact with the data. When you have a program it usually takes some sort of data into it to solve a specific problem. In the case of automation we take in data structures such as the Junos config and then use that data in various ways. We may manipulate the data or config, provide a report or output the data, or even remove elements and commit the change back to the Junos device.

Variables

A variable is a named data structure that contains information used by the program. There are several type of variables that can be used in Python. All Python data structures are known as objects. An object is a data structure that not only contains data but it also contains methods. A method gives the object certain actions that it can take. We will take a more detailed look at this in the next section.

Numbers

A number in Python is an integer that can contain either a positive or negative value.

Python has four different types of numbers that you can use. This varies depending on how large the number is or how precise the number needs to be.

- int
    - A signed integer that can be a positive or negative number

    - 1 or -1 are both ints
- long
    - A large number specified with a capital L at the end of the number
    - 51924361L or -51924361L
- float
    - A number with floating points specified with a decimal point and a following set of integers
    - 1.01 or -20.2
- complex
    - A number that is very large or complex and is followed with a lower case j
    - 3.14j

Strings

A string is a series of characters. This can include letters or numbers. All Python strings are encoded in UTF-8 as opposed to ASCII. This lets users utilize non-English characters and ultimately simplifies dealing with text.

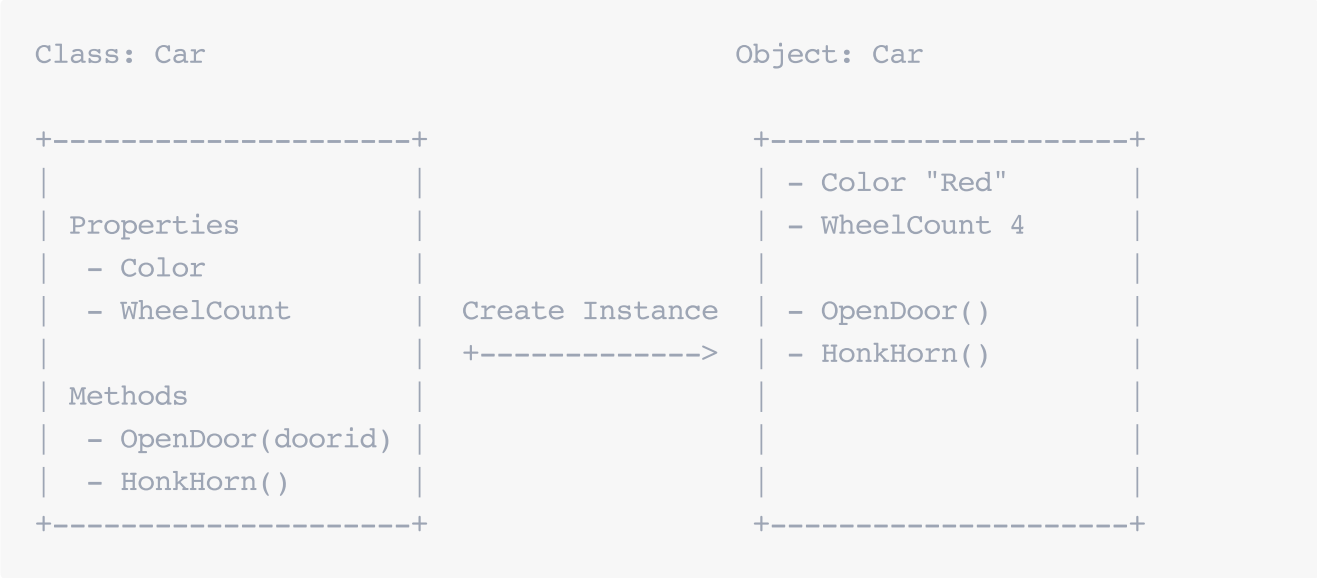
Lists and Tuples

A list or tuple is a series of data. They can contain any type of data and the types within a list can be mixed. So you could have an object, a number, and a string contained in either. The list and tuple is the same, except for that a tuple can not be manipulated and is immutable. A list can be reordered or have items deleted from it as needed.

Classes and Objects

A class is a complex data structure. It contains variables or properties as well as methods. A method is a function that can be used as actions on the object. The classic example for class is to think of it as a blueprint to create an object. A car has certain properties such as color or number of wheels. If you wanted to open a car door that would be a method or action that the object could take. Object oriented programming was invented by Alan Kay at Xerox PARC in the 1970s as a way of coupling data with methods of operation. Classes and object-oriented programming is quite a tall subject we won't go into all the details here.

Classes and Objects Examples



Object Oriented Programming

- Object Oriented Programming Defined
- List of OOP Languages
- OOP Design Patterns
  - This book is written by the "Gang of Four" and considered the ultimate guide to OOP design patterns
- Multi-paradigm Languages

Examples of variables

```
#number
simple_int = 1

#string
simple_str = "foo"

#a list of ordered objects
simple_list = ["foo","bar",1]

#an tuple is an immutable list
simple_tuples = ("foo", "bar", 1)

#a dict or dictionary is a key/value structure
simple_dict = {'foo':{'bar':True},'shoe':1}

# a class is a structure that defines data storage and operations on the da
class Calculator:
    foo = 0
    bar = 0
    def __init__(self,foo,bar):
        foo = foo
        bar = bar

    def add(self):
        return foo + bar

simple_object = SimpleClass(1,2)
simple_return = simple_object.add()

print simple_return
# Prints 3
```

Functions and Methods

As discussed a function or method is a bit of code that can be repeatedly reused. This is typical for a common set of code that we would use several times. By making this into a function or method it leaves the code in a singular place in which it can be reused but also maintained. If you were to have a function to add numbers, as can be seen below in our example "add_numbers", we can call this function any time we need to return the sum of two numbers. If we want to change this behavior in the code we can update this single function and then have that change impact the code within the rest of the program.

A method is simply a function that is bound to an object or class. This is usually used to describe the actions or methods of operation over a class. They are written nearly identically. However when defining a method on a class the special object "self" must be the first parameter used. This allows the object to reference itself and change internal elements to its own data structure. This means you are changing the instantiated object, rather than changing the class all together.

```
# Function
def add_numbers(i1, i2):
    return i1 + i2

# Method is a function bound to an object
class Calculator:
    def add(self, i1, i2):
        return i1 + i2
```

Conditionals

A conditional is a basic set of logic in programming. The most common here is the "if" statement.

```
foo = True
bar = False

# a simple if statement
if foo:
    print "success"

# a simple if else statement
if foo:
    print "success"
else:
    print "fail"

# a simple if else if and else statement
if foo != True:
    print "success foo"
elif bar != True:
    print "success bar"
else:
    print "fail"
```

Loops

A loop is the ability to loop over a data structure. This lets you take a structure and act upon each element. This is also known as iteration.

A "for" loop takes a variable and iterates over it. It provides one value from the list and allows you to utilize it. In the example below we want to take one of each of the items and then print it out. In our example we take a list of four alcoholic drinks and then have it request an order for one.

Also you can use a "while" loop. A "while" loop continues to loop until the while condition changes. In this example we are saying "while true" to continue the loop. This loop will run indefinitely unless we choose to break from the loop. In our infinite loop we increment the value of x by one. After 11 iterations, since we start at 0, we detect that x is equal to 10. Once this is detected we print out the current value of x and declare we are complete. Lastly we call the command break to exit the execution of the loop.

```
# a simple for loop
items = ["gimlet","greyhound","old fashioned","beer"]

for item in items:
    print "Order me a {0} please!".format(item)

# Prints
# Order me a gimlet please!
# Order me a greyhound please!
# Order me a old fashioned please!
# Order me a beer please!

# a simple while loop
x = 0

while True:
    if x == 10:
        print "Value of X is {0}".format(x)
        print "Complete"
        break
    x = x + 1
```

Python Summary

Learning programming or a specific language is much like playing the game of golf. It is something you can do for years to get better and better at it. Experience in using Python will be your greatest advantage to getting better at it. Find use cases for it in your daily life. You can simply drop into an interactive Python shell, just type "python" at your prompt and go for it, and use it. Use it as a calculator, use it to read in a spreadsheet and iterate over the data, use it to automate your network. What at first seems impossible, becomes completely possible, and eventually it becomes natural. Only by using it and solving new problems will you see growth in your skill sets.

- What is Python
- Python Site
- Python for Network Engineers
- Learn Python the Hard Way

Getting Started with PyEZ

The PyEZ library is a simple way to utilize Python to interact with Junos. It uses some special programming techniques that keep it free from needing to be updated to support new RPC calls in Junos. It takes a complex process of connecting to the device, running RPCs, and then parsing the output and makes it a snap. However just because the name includes EZ for easy this does not inheritly make it easy to use. There are some important basics that you would need to know before you get started.

Metaprogramming

Metaprogramming is a concept that is applicable in several programming languages. It effectively allows you to call code that does not exist and it utilizes the program as the data. This means that the program can dynamically interpet the code that is called and turn it into other code for you. The feelings around this technique vary, but in the case of PyEZ it is a perfect use case. The downside to consider is that this code is dynamically generated with no specific documentation around the various available methods on the device. Developers tend to spend a lot of time reading documentation and because the code doesn't actually exist this can be a problem. However the target of the PyEZ library is that it is to be used by network engineers who cross the lines between Junos knowledge and Python. This ability to meet in the middle is very powerful in the right hands.

In Junos there are potentially thousands of different RPC calls that the user can call. These RPC calls are well documented on the [Juniper website]TODO undate link (http://foo). This documentation tells you what the various RPC calls do and the expected responses.

Mapping RPCs to Methods - The method to the madness

As someone familiar with Junos you have a distinct advantage over the average programmer. As you know, or at least know how, to find the RPCs that you are looking to call. There is a mapping between the RPC calls. If you know the command or can figure it out it is simple to learn the correct RPC call. You can use the format of "show interfaces terse ge-0/0/0 | display xml rpc". This will show you what you need to use to call the command in PyEZ.

RPC Mapping Examples

```
# Command "show interfaces terse ge-0/0/0"
# RPC - get-interface-information
# PyEZ - get_interface_information

# Command on the CLI
root@NetDevOps-SRX01> show interfaces terse ge-0/0/0
Interface               Admin Link Proto    Local                 Remote
ge-0/0/0                up    up
ge-0/0/0.0              up    up   inet     10.0.2.15/24

root@NetDevOps-SRX01>

# Show XML output from the CLI
root@NetDevOps-SRX01> show interfaces terse ge-0/0/0 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <interface-information xmlns="http://xml.juniper.net/junos/12.1X47/juno
        <physical-interface>
            <name>ge-0/0/0</name>
            <admin-status>up</admin-status>
            <oper-status>up</oper-status>
            <logical-interface>
                <name>ge-0/0/0.0</name>
                <admin-status>up</admin-status>
                <oper-status>up</oper-status>
                <filter-information>
                </filter-information>
                <address-family>
                    <address-family-name>inet</address-family-name>
                    <interface-address>
                        <ifa-local junos:emit="emit">10.0.2.15/24</ifa-loca
                    </interface-address>
                </address-family>
            </logical-interface>
        </physical-interface>
    </interface-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>

#show XML RPC from the command line
root@NetDevOps-SRX01> show interfaces terse ge-0/0/0 | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <rpc>
        <get-interface-information>
                <terse/>
                <interface-name>ge-0/0/0</interface-name>
        </get-interface-information>
```

```
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>

root@NetDevOps-SRX01>
```

PyEZ Simple Example

In this example:

- Initialize device object
- Open a connection to the device
- Print the facts
- Close the connection

```
from jnpr.junos import Device

dev = Device( user='root', host='172.16.0.1', password='Juniper' )

dev.open()

print dev.facts

dev.close()
```

PyEZ Example with RPC

- Initialize device object
- Open a connection to the device
- Request the chassis inventory
- Print the text representation of the XML
- Close the connection

```
from jnpr.junos import Device
import xml.etree.ElementTree as ET

junos_dev = Device(user='root', host='172.16.0.1', password='Juniper' )

junos_dev.open()

chassis_output = junos_dev.rpc.get_chassis_inventory()

print ET.tostring(chassis_output,encoding="utf8", method="text")

junos_dev.close()
```

Tables and Views

As we have seen previously, Junos maintains all output data in an XML structure that allows for easy processing. While we could use standard tools like Regular Expressions, or more likely the very powerful XPath tools, for translating these XML elements into Python variables, we have simplified this process.

For the many use cases for collecitng data from a Junos device we follow the same process:

1. Send RPC to device
2. Receive XML
3. Parse XML and loop through data

To parse this data from XML nested information, into useful Python data objects, the Juniper team created two primary components:

- **Tables:** A table essentially represents all the data collected for a certain RPC request, sorted and keyed on a particular set of values and presented to the user as a collection of native Python data structures. The end user only needs to describe the data sets in YAML format without knowing the Python objects below.
    - Think about Tables as defining the data set and the rows of a table
- **Views:** A view is applied to a table to create a custom combination of the data in the table. As such, a combination of table and view definitions can be created with just a handful of lines of code yet scale infinitely.
    - Think about Views as defining the columns of a table

These two features are very powerful, but are somewhat abstract concepts. To better

demonstrate the way tables and views work together it's best to show you a real world example.

In this example, we're going to build on the XML examples shown previously in the "RPC Mapping Examples" section, and show how we can use PyEZ to programmatically collect details about port states. This might feel like a simplistic use case, but this is actually the script that the author first wrote when starting out with the PyEZ library to solve a real world "on the job" customer problem.

The first thing we need to do is map the <rpc> for the "show interface" command to a table structure. A table takes all of the XML output and collects into an ordered fashion. Table and view definitions are written in the YAML format, which is an easy to read format that is also programming language independent. The PyEZ library stores these definitions in the "lib/jnpr/junos/op/" directory in the folder where the PyEZ library is stored.

First, lets have a look at the following table definition that ships with the PyEZ library.

PhyPort Table Definition:

```
PhyPortTable:
  rpc: get-interface-information
  args:
    interface_name: '[fgx]e*'
  args_key: interface_name
  item: physical-interface
  view: PhyPortView
```

If you recall from our investigations in the "DMI and the NETCONF Protocol" the RPC command associated with "show interface" command is <get-interface-information>. The second line of the output above requests that this table definition should map to the XML output from that <rpc-reply>. We then filter down our interface list to just Fast Ethernet, Gigabit Ethernet and 10 Gigabit Ethernet interfaces ("[fge]e*"). The "item" section creates the equivalent of the table row in this data structure. By defining "physical-interface" as the "item" each interface from the XML output will be represented separately in the Table Structure.

The last line in the output above is used to feed this table into a "view". The view definition below takes each of the XML tags from the <rpc-reply> and maps them to dictionary keys on the left. These keys are the equivalent of columns in this data structure.

PhyPort View Definition:

```
PhyPortView:
  fields:
    oper : oper-status
    admin : admin-status
    description: description
    mtu: { mtu : int }
    link_mode: link-mode
    speed: speed
    macaddr: current-physical-address
    flapped: interface-flapped
```

With these Tables and Views defined, we can now write some very simple Python code to connect to our switch and collect interface statistics using these definitions. The example file below is a very simple script designed to loop through all the Ethernet interfaces on the switch and print out a “CSV formatted” output of the port statistics on the switch.

port-report.py:

```
from jnpr.junos.op.phyport import *
from jnpr.junos import Device

dev = Device( user='netconf-test', host='lab-switch', password='lab123' )
dev.open()

ports = PhyPortTable(dev).get()
print "Port,Status,Flapped" #Print Header for CSV

for port in ports:
        print("%s,%s,%s" % (port.key, port.oper, port.flapped))
```

The Python code show here is quite simple, and we have only introduced three new components from our helloEZ.py example.

- In the first line we load the library module for the PhyPort table and view. This is required to ensure that our script knows where to find our Table and View definitions.
- After “dev.open()” we call our module to with the “PhyPortTable(dev).get()”. We store the result of this function in the variable “ports”. Once assigned, this variable now contains the result of the Table+View combination.
    - This is represented in Python as a List structure, where each item is a Python

Dictionary (Collection of Key/Value pairs).

- Now that we have the data from our switch represented in simple iterable Python objects we can process them using standard Python tools. To present the output to the use, we then loop through our table printing out the key (which happens to be our interface name), the Operational State of the port, and the last time the port changed state.

When we run this script we get the following output:

```
vagrant@NetDevOps-Student:/vagrant/examples$ python ./port-report.py
Port,Status,Flapped
ge-0/0/0,up,2015-03-30 21:30:40 UTC (02:05:54 ago)
ge-0/0/1,up,2015-03-30 21:30:40 UTC (02:05:54 ago)
ge-0/0/2,up,2015-03-30 21:30:40 UTC (02:05:54 ago)
vagrant@NetDevOps-Student:/vagrant/examples$
```

This example output shows that we now have a simple Comma Delimited list containing the port name, current operational status and how long since the port changed status for each port on our switch. If you wanted to earn bonus points with your boss, it is trivial to take the CSV output and put pretty formatting around it in Excel to present management style reports. Extra bonus points if you create the reports using all Python!

This was just a simple example of automating the operational tasks on a Junos platform, but by combining the power of the Junos DMI system with the NETCONF protocol and the Tables and Views structures from the PyEZ library, we can easily automate many of our troubleshooting and data collection tasks.

Templating with Jinja2

In Python the most popular templating language is called Jinja. The original use case for Jinja was to provide developers a simpler method for developing web applications. Today most of the HTML pages that you see are actually created via templating. In the use case of of Junos configurations we most likely do not need all of the features of Jinja.

The complete documentation can be found here Jinja Dev Documentation. However we will highlight the specific language use cases here.

Utilized Jinja features

Jinja has a huge set of features that can be utilized to accomplish your goals. But the hardest part in getting started is learning where to start. For this demo we will use only a small subset of the Jinja template engine.

Variable Substitution

The most basic example of Jinja is variable substitution. Think of this as a simple search and replace within a text editor or using the UNIX command *sed*. There are several benefits to utilizing a Python script to accomplish this task.

- Repeatability
  - The ability to repeatedly call a script with fixed input and specified output
- Chaining
  - The ability to chain scripts and or actions together

To create a Jinja template it is quite simple. Below is an example of a Jinja substitution variable.

Hello World Template

```
"Hello, {{ world }}!"
```

In our example you see "Hello, " which is just treated as plain text. When run through the Jinja engine this text will be output as is. What Jinja is interested in is the contents of the double curly braces "{{ world }}". Jinja sees this as a variable named "world". When applying a variable you are simply applying a key and value to the variable. In this case "world" is the variable. In the Python code we would simply apply world="Automation". The message outputted would be "Hello, Automation!" if you applied this simple key and value.

Here is the full example of what we just discussed.

Full Hello World Template

```
# Template
"Hello, {{ world }}!"

# Key/Value
world="Automation"

# Final Output
"Hello, Automation!"
```

The documentation for this can be found here.

Using loops

The second step in substitution is utilizing loops. Loops provide true automation advantage to not require the user to repetitively type in commands. We can apply a more complex data structure in Python to a template and iterate over the items. To do this we use the age old "for loop". A "for" loop is used in many different languages to loop or iterate over items.

A Jinja for loop example:

```
{% for item in items %}
    Hello, {{ item }}!
{% endfor %}
```

In this example we use an array or list called *items*. This contains a list of strings. When run Jinja will take each item out of the list *items*. A list can contain mixed types or different types of information. The same list could contain numbers/integers, strings, or even more complex items such as dictionaries. The loop takes the next item in the list, starting at item 0, and substitutes it in the loop.

Full for loop example

```
# Template
{% for item in items %}
    Hello, {{ item }}!
{% endfor %}

# Key/Value
items = ["Automation","Shoe","Socks","Pants"]

# Final Output
Hello, Automation!
Hello, Shoe!
Hello, Socks!
Hello, Pants!
```

Best practices in writing tools

While there is a mountain of wisdom that you can use in programming, here are a few best practices in writing tools that are critical to think about.

1. Keep it simple
    - Don't try and cram too many goals into a single tool
2. Keep tools to a singular workflow
    - The best automation is a set of loosely coupled tools that are strung together
    - This simplifies the debugging and maintenance of the tools
    - Chaining multiple workflows together is called orchestration
    - Use an orchestration tool to chain the tools together
3. Think about the consumer of the tool
    - Can they use it easily
    - Are there a lot of dependencies to run it
    - Does the tool solve the problem of the user
    - Will they need your help after you give them the tool
    - How will they get access to the tool
        - github, package distribution, email, website
4. Maintainability
    - Can the tool be easily maintained
    - Are there other use cases for the tool
    - Does it have clear documentation on how to use it