Getting Started with Python for Network Engineers

Module Overview

- Python Interpreter
- Data Types
- Using and Creating Python Libraries
- Working with Files
- Writing Python Scripts
- Conditionals
- Loops
- Functions

Python Interactive Interpreter

- Often called the "Python Shell"
- Used for rapid prototyping and testing
- Does not require the user to create scripts, programs, etc.
- No text editor or IDE required
- Just type "python" within a terminal session (Linux/MAC)

```
ntc@jump-host:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>> # ENTER PYTHON CODE / STATEMENTS HERE
```

Data Types - Strings

Introduction to Python for Network Engineers

Strings

- Sequence of characters enclosed by quotes
- Single or double quotes are accepted, but be consistent
- Task:
  - Create a variable called `hostname` and assign it the value of "ROUTER1"

```
netdev@networktocode:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> hostname = "ROUTER1"
```

Strings

- Sequence of characters enclosed by quotes
- Single or double quotes are accepted, but be consistent
- Task:
  - Create a variable called `hostname` and assign it the value of "ROUTER1"

```
netdev@networktocode:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> hostname = "ROUTER1"
```

- Task:
  - Create a variable called `ip_addr` and assign it the value of "10.200.1.1/24"

```
>>> ip_addr = "10.200.1.1/24"
>>>
```

type() Function

Verify the type of object using the `type()` function.

```
>>> ip_addr = "10.200.1.1/24"
>>>
>>> type(ip_addr)
<type 'str'>
>>>
```

```
>>> hostname = 'r1'
>>>
>>> type(hostname)
<type 'str'>
>>>
```

`str` denotes the object is a String

Printing Strings

- With the `print` statement:

```
>>> ip_addr = "10.1.100.1"
>>> print(ip_addr)
10.1.100.1
>>>
```

```
>>> ip_addr
'10.1.100.1'
>>>
```

Printing Strings

- With the `print` statement:

```
>>> ip_addr = "10.1.100.1"
>>> print(ip_addr)
10.1.100.1
>>>
```

```
>>> ip_addr
'10.1.100.1'
>>>
```

- With vs. without the `print` statement:

```
>>> ip_addr = "The IP Address is:\n\n10.1.100.1"
>>>
>>> ip_addr
'The IP Address is:\n\n10.1.100.1'
```

```
>>> print(ip_addr)
The IP Address is:

10.1.100.1
```

Concatenation

- "Add" or *concatenate* strings

```
>>> ip_addr = '10.1.100.1'
>>> ipmask = ip_addr + '/24'
>>>
>>> print(ipmask)
10.1.100.1/24
>>>
```

- Without pre-built variables...

```
>>> print("The IP Address and Mask is: " + ipmask)
The IP Address and Mask is: 10.1.100.1/24
>>>
>>> statement = "The IP Address and Mask is: " + ipmask
>>> statement
'The IP Address and Mask is: 10.1.100.1/24'
>>>
```

Printing Repeating Strings

- Using the asterisk as a mathematical multiplier
- Handy when trying to format output.

```
>>> hyphen = '-'
>>>
>>> print(hyphen * 30)
------------------------------
>>>
```

```
>>> print('123' * 10)
123123123123123123123123123123
>>>
```

Built-in String Methods

- Python, like other languages, offers built-in ways to work with and manipulate strings.
- Use `dir()` on any variable to see built-in methods available

```
>>> # verify data type with type()
>>>
>>> type(host1_ip)
<type 'str'>
>>>
```

```
>>> # verify built-in methods with dir()
>>>
>>> host1_ip = '10.100.1.1'
>>>
>>> dir(host1_ip)
['capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
>>>
>>>
>>> # cleaned up for brevity
```

Built-in String Methods (cont'd)

Use the object (variable) or data type name with the `dir()` function.

- Strings - str
- Lists - list
- Dictionaries - dict
- Sets - set
- Tuples - tuple

```
>>> dir(str)
['capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
>>>
>>>
>>> # cleaned up for brevity
```

Built-in String Methods (cont'd)

- Ways to simplify working with and performing common operations with strings
- Built-in methods (and attributes) exist for all objects, not just strings or Python standard data types
- Key questions:
  - What data is being returned (if any)?
  - What data type is the data being returned?
  - Is the original object (variable) modified?

startswith()

- Checks to see if the string starts with certain char(s)
- Returns: `boolean`
- Task: Check to see if the first octet is equal to 10

```
>>> host1_ip = '10.100.1.1'
>>>
>>> host1_ip.startswith('10')
True
```

- Store the result and print it

```
>>> ip_check = host1_ip.startswith('10')
>>>
>>> print(ip_check)
True
```

help() Function

Get built-in help statements on how to use a given method.

```
>>> help(host1_ip.startswith)
>>>
```

```
Help on built-in function startswith:

startswith(...)
    S.startswith(prefix[, start[, end]]) -> bool

    Return True if S starts with the specified prefix, False otherwise.
    With optional start, test S beginning at that position.
    With optional end, stop comparing S at that position.
    prefix can also be a tuple of strings to try.
(END)
```

Learning Workflow

- Verify data type
  - `type(host1_ip)`
- View built-in methods and attributes
  - `dir(host1_ip)`
- Learn about built-in methods and attributes
  - `help(host1_ip.startswith)`

upper() and lower()

- Modifies the case of a string
- Returns: `string`
- Task:
  - Use the lower method to convert a string to all lower case letters

```
>>> hostname = 'NYCWAN1'
>>>
>>> hostname.lower()
'nycwan1'
>>>
>>> hostname
'NYCWAN1'
```

upper() and lower() - (cont'd)

- Using help()

```
>>> help(str.upper)          # or help(hostname.lower) from previous example
```

```
Help on method_descriptor:

upper(...)
    S.upper() -> string

    Return a copy of the string S converted to uppercase.
(END)
```

replace()

- Replaces character(s) in a string
- Returns: `string`
- Task:
  - Ensure consistency for the format of MAC addresses
- Possible formats include:
  - aa:bb:cc:dd:ee:ff
  - aa-bb-cc-dd-ee-ff
  - aa.bb.cc.dd.ee.ff

replace()

- Replaces character(s) in a string
- Returns: `string`
- Task:
  - Ensure consistency for the format of MAC addresses
- Possible formats include:
  - aa:bb:cc:dd:ee:ff
  - aa-bb-cc-dd-ee-ff
  - aa.bb.cc.dd.ee.ff
- Replace all `.` 's with `:` 's

```
>>> mac_addr = 'aa.bb.cc.dd.ee.ff'
>>>
>>> new_mac = mac_addr.replace('.',':')
>>>
>>> print(new_mac)
aa:bb:cc:dd:ee:ff
>>>
```

split()

- Split a string into a list based on a char(s)
- Splits based on whitespace by default
- Returns: `list`
- Task:
  - Print the first octet of an IP Address

```
>>> ipaddr = '192.168.100.50'
>>>
>>> ipaddr.split('.')
['192', '168', '100', '50']
>>>
```

split()

- Split a string into a list based on a char(s)
- Splits based on whitespace by default
- Returns: `list`
- Task:
  - Print the first octet of an IP Address

```
>>> ipaddr = '192.168.100.50'
>>>
>>> ipaddr.split('.')
['192', '168', '100', '50']
>>>
```

```
>>> ipaddr_list = ipaddr.split('.')
>>>
>>> first_octet = ipaddr_list[0]
>>>
>>> print(first_octet)
192
```

splitlines()

- Similar to `split`, but works on line breaks (`\n`) and carriage returns (`\r`)
- `str.split('\n')` is *nearly* equivalent to `str.splitlines()`
- Returns: `list`

```
>>> show_run_interface = 'interface Ethernet1/1\n  no switchport\n  description Python Testing\n  ip address 10.1.1.1/24'
>>>
>>> print(show_run_interface)
interface Ethernet1/1
  no switchport
  description Python Testing
  ip address 10.1.1.1/24
>>>
```

```
>>> lines = show_run_interface.splitlines()
>>> lines
['interface Ethernet1/1', '  no switchport', '  description Python Testing', '  ip address 10.1.1.1/24']
>>>
```

strip(), lstrip(), rstrip()

Remove leading and trailing white space

```
>>> command = "    show run "
>>>
>>> command.strip()
'show run'
>>>
```

Remove leading white space

```
>>> command.lstrip()
'show run '
>>>
```

Remove trailing white space

```
>>> command.rstrip()
'    show run'
>>>
```

join()

- Join together characters or elements in a list inserting a char(s) between each
- Returns: `string`
- Task: Create a command string from a list of commands

```
>>> commands = ['interface Ethernet1/1', 'switchport access vlan 10', 'no shut']
>>>
>>> type(commands)
<type 'list'>
>>>
```

```
>>> ' ; '.join(commands)
'interface Ethernet1/1 ; switchport access vlan 10 ; no shut'
>>>
```

```
>>> command_string = ' ; '.join(commands)
>>> print(command_string)
interface Ethernet1/1 ; switchport access vlan 10 ; no shut
>>> type(command_string)
<type 'str'>
>>>
```

join() - cont'd

- Be aware of your data types
- join() works with strings too

```
>>> command = 'switchport mode access'
>>>
>>> ' ; '.join(command)
's ; w ; i ; t ; c ; h ; p ; o ; r ; t ;   ; m ; o ; d ; e ;   ; a ; c ; c ; e ; s ; s'
>>>
```

Note: for our use cases, this will be used more with lists, but showing for completeness

len()

- Obtain the length of strings, lists, and other data types
- Returns: `integer`
- Functions vs. Methods
  - Can it be called directly?

```
>>> hostname = 'DCNJSWITCH1'
>>>
>>> len(hostname)
11
```

```
>>> commands = ['interface Ethernet1/1', 'switchport access vlan 10']
>>>
>>> len(commands)
2
>>>
```

format()

- Perform variable substitution for string formatting
- Returns: `string`

```
>>> hostname = 'nxos-spine1'
>>> vendor = 'cisco'
>>>
>>> print('Device hostname is {}, shipped from {}'.format(hostname, vendor))
Device hostname is nxos-spine1, shipped from cisco
```

Later versions of Python do not require the numbers in curly braces:

```
>>> ip = '10.1.1.1'
>>> mask = '24'
>>>
>>> ip_addr_command = 'ip address {}/{}'.format(ip, mask)
>>>
>>> print(ip_addr_command)
ip address 10.1.1.1/24
>>>
```

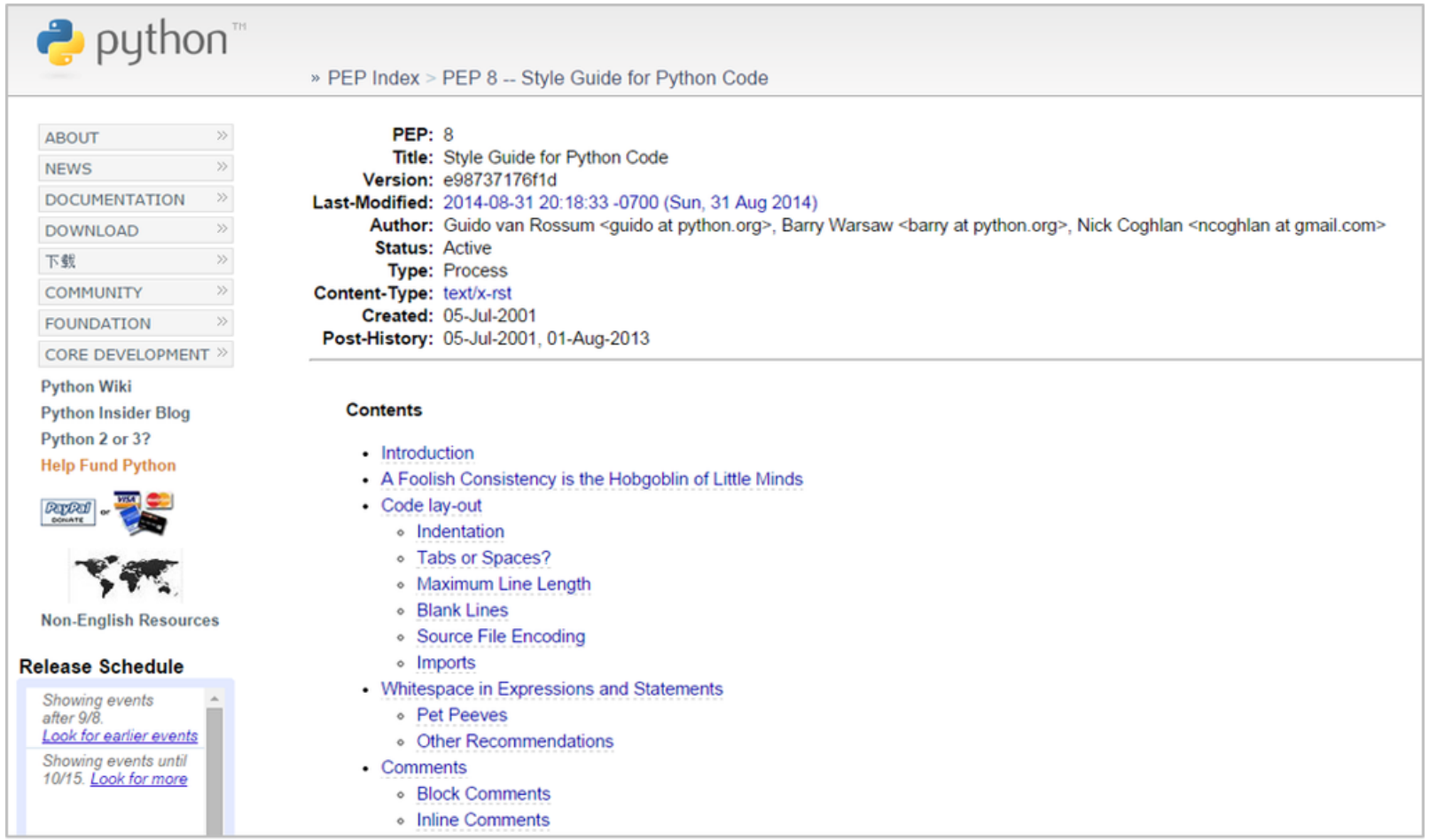
Before the First Lab - Things to Know

White Space

- Generally the amount of space is not that important
- Just be consistent
- Tabs vs. Spaces
- PEP8
- The examples below work, but are not good habits to form
  - `hostname = router1` is a good quality example

```
>>> hostname='router1'
>>>
>>> hostname       =           'router1'
>>>
```

Style - PEP8



python™

» PEP Index > PEP 8 -- Style Guide for Python Code

ABOUT
NEWS
DOCUMENTATION
DOWNLOAD
下载
COMMUNITY
FOUNDATION
CORE DEVELOPMENT

Python Wiki
Python Insider Blog
Python 2 or 3?
Help Fund Python

Non-English Resources

Release Schedule

Showing events after 9/8.
Look for earlier events
Showing events until 10/15. *Look for more*

PEP: 8
Title: Style Guide for Python Code
Version: e98737176f1d
Last-Modified: 2014-08-31 20:18:33 -0700 (Sun, 31 Aug 2014)
Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status: Active
Type: Process
Content-Type: text/x-rst
Created: 05-Jul-2001
Post-History: 05-Jul-2001, 01-Aug-2013

Contents

- Introduction
- A Foolish Consistency is the Hobgoblin of Little Minds
- Code lay-out
  - Indentation
  - Tabs or Spaces?
  - Maximum Line Length
  - Blank Lines
  - Source File Encoding
  - Imports
- Whitespace in Expressions and Statements
  - Pet Peeves
  - Other Recommendations
- Comments
  - Block Comments
  - Inline Comments

http://legacy.python.org/dev/peps/pep-0008/

Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Python 2.x vs Python 3.x

Python3 removed some inefficiencies (like better memory management for lists and dictionaries) in addition to removing bugs and adding new features.

Some significant differences for the beginner:

- `print` is a function, not a statement
- `raw_input` in Python2 is `input` in Python3
- `3/4 = 0` in Python 2 vs `3/4 = 0.75` in Python 3
- Improvements with error messages while calling functions with incorrect arguments

The main driver for Python3 was to correctly handle unicode. In Python2 the default is byte encoding for strings (using ASCII encoding); in Python3 the default is unicode. This lead to a lot of bugs while handling non-english characters and symbols.

How does Python2 vs Python3 affect network automation

- Most 3rd party libraries that are needed for network automation have been ported to support Python3
- Some exceptions may be encountered such as :
  - `pyntc` has dependencies on the `textfsm` library. The Python3 compliant `textfsm` has to be manually installed for the `pyntc` installation to work.

Summary

- Use the Python Interpreter...often
- `type()` , `dir()` , `help()`
- Built-in methods exist for every data type
  - You must understand how to use them
  - What data gets returned?
  - Does the original object get modified?

Lab Time

- Lab 1 - Accessing the Lab Environment
  - Learn how to access the lab environment
- Lab 2 - Using the Python Interpreter
  - Getting Started with the Python Interactive Interpreter
- Lab 3 - Working with Strings
  - Learn how to work with strings and their built-in methods while working in the Python Interactive Interpreter

Accessing Lab Guide Use the `LAB_GUIDE.md` file in the root of the GitHub repository.

Data Types - Numbers

Introduction to Python for Network Engineers

Numbers - integers

- Integers
  - Whole numbers such as 1, 2, 3, 4, 100.
  - Those that do not have a decimal point
- You can use mathematical operators directly within the Python shell

```
>>> 5 * 4
20
>>> x = 5
>>> y = 10
>>>
>>> x * y
50
>>>
```

```
>>> z = 3
>>> y / z
3
>>>
```

Numbers - floats

- More precise than integers
- Decimal points are used
- At least one number must be a float

```
>>> z = 3.0
>>> y = 10
>>>
>>> result = y / z
>>> result
3.3333333333333335
>>>
>>> round(result, 2)
3.33
>>>
```

Data Types - Lists

Introduction to Python for Network Engineers

Lists

- Store multiple objects as an **ordered** list
- Indexed by an integer value starting at 0
- Sometimes referred to as arrays
- Elements can be of different data types

Example:

```
>>> commands = ['interface Ethernet1/1', 'switchport access vlan 10']
```

```
>>> commands[0]
'interface Ethernet1/1'
>>>
```

```
>>> commands[1]
'switchport access vlan 10'
>>>
```

Lists (cont'd)

- Store and access data about network devices

```
>>> device1 = ['switch1', '10.1.100.1/24', '00:00:00:00:00:01', 'cisco']
>>>
>>> device2 = ['switch2', '10.1.100.2/24', '00:00:00:00:00:02', 'arista']
>>>
>>> mac_1 = device1[2]
>>> mac_2 = device2[2]
>>>
>>> vendor_1 = device1[3]
>>> vendor_2 = device2[3]
>>>
>>> print(mac_1, mac_2)                    # print multiple vars on same line
00:00:00:00:00:01 00:00:00:00:00:02
>>>
>>> print(vendor_1, vendor_2)
cisco arista
>>>
```

Note: you should not store contextual data like this in lists. We will cover why later.

Built-in List Methods

- Just like strings, lists have several methods that can be used to manipulate and simplify working with them.

```
>>> device1 = ['switch1', '10.1.100.1/24', '00:00:00:00:00:01', 'cisco']
>>>
>>> dir(device1)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
>>>
>>> # cleaned up for brevity
```

Built-in List Methods

- Just like strings, lists have several methods that can be used to manipulate and simplify working with them.

```
>>> device1 = ['switch1', '10.1.100.1/24', '00:00:00:00:00:01', 'cisco']
>>>
>>> dir(device1)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
>>>
>>> # cleaned up for brevity
```

- You can also use the data type to examine the list of built-in methods, i.e. str, list, etc.

```
>>> dir(list)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
>>>
>>> # cleaned up for brevity
```

append()

- Add or *append* element to a list
- Returns: nothing; modifies existing list
- Task:
  - Add switch model to a list that holds device characteristics (facts)

```
>>> device1.append('nexus_9396')
>>>
>>> device1
['switch1', '10.1.100.1/24', '00:00:00:00:00:01', 'cisco', 'nexus_9396']
>>>
```

insert()

- Inserts new value into a specific location (element) in a list
- Returns: nothing; modifies existing list

```
>>> commands = ['interface Ethernet1/1', 'switchport access vlan 10']
```

- Tasks:
  - Add "config t" to the list as the first element
  - Add "switchport mode access" in front of the "switch access vlan" command (3rd element)

```
>>> commands.insert(0, 'config t')
>>>
>>> commands
['config t', 'interface Ethernet1/1', 'switchport access vlan 10']
```

insert()

- Inserts new value into a specific location (element) in a list
- Returns: nothing; modifies existing list

```
>>> commands = ['interface Ethernet1/1', 'switchport access vlan 10']
```

- Tasks:
  - Add "config t" to the list as the first element
  - Add "switchport mode access" in front of the "switch access vlan" command (3rd element)

```
>>> commands.insert(0, 'config t')
>>>
>>> commands
['config t', 'interface Ethernet1/1', 'switchport access vlan 10']
```

```
>>> commands.insert(2, 'switchport mode access')
>>>
>>> commands
['config t', 'interface Ethernet1/1', 'switchport mode access', 'switchport access vlan 10']
```

pop()

- By default, `pop` removes the last element
- Returns: element being removed; original list is also modified
- Task:
  - Remove the device model just added

```
>>> device1
['switch1', '10.1.100.1/24', '00:00:00:00:00:01', 'cisco', 'nexus_9396']
>>>
>>> device1.pop()
'nexus_9396'
```

pop()

- By default, `pop` removes the last element
- Returns: element being removed; original list is also modified
- Task:
  - Remove the device model just added

```
>>> device1
['switch1', '10.1.100.1/24', '00:00:00:00:00:01', 'cisco', 'nexus_9396']
>>>
>>> device1.pop()
'nexus_9396'
```

```
>>>
>>> device1
['switch1', '10.1.100.1/24', '00:00:00:00:00:01', 'cisco']
>>>
```

pop() - (cont'd)

- Task:
  - Remove the IP address from the list

```
>>> device1
['switch1', '10.1.100.1/24', '00:00:00:00:00:01', 'cisco']
```

```
>>> device1.pop(1)          # ip address is 2nd element w/ index of 1
'10.1.100.1/24'
```

```
>>> device1
['switch1', '00:00:00:00:00:01', 'cisco']
```

pop() - (cont'd)

- Using help

```
>>> help(list.pop)
Help on method_descriptor:

pop(...)
    L.pop([index]) -> item -- remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.
(END)
```

count()

- How many of the same element exists within a list?
- Returns: `integer`
- Task:
  - Identify how many of the switches are Cisco switches

```
>>> switches = ['cisco', 'cisco', 'arista', 'cumulus', 'cumulus', 'cisco']
>>>
>>> switches.count('cisco')
3
>>>
```

extend()

- Combine two lists
- Returns: nothing; updates original list
- Task:
  - *Extend* the production IP addresses with those coming in from QA/Test

```
>>> prod = ['10.1.1.1', '10.1.1.5', '10.1.1.9']
>>>
>>> qatest_ip = ['10.1.1.8', '192.168.1.5', '192.168.1.8']
>>>
>>> prod.extend(qatest_ip)
>>>
>>> prod
['10.1.1.1', '10.1.1.5', '10.1.1.9', '10.1.1.8', '192.168.1.5', '192.168.1.8']
```

extend()

- Combine two lists
- Returns: nothing; updates original list
- Task:
  - *Extend* the production IP addresses with those coming in from QA/Test

```
>>> prod = ['10.1.1.1', '10.1.1.5', '10.1.1.9']
>>>
>>> qatest_ip = ['10.1.1.8', '192.168.1.5', '192.168.1.8']
>>>
>>> prod.extend(qatest_ip)
>>>
>>> prod
['10.1.1.1', '10.1.1.5', '10.1.1.9', '10.1.1.8', '192.168.1.5', '192.168.1.8']
```

```
>>> prod = ['10.1.1.1', '10.1.1.5', '10.1.1.9']
>>>
>>> qatest_ip = ['10.1.1.8', '192.168.1.5', '192.168.1.8']
>>>
>>> prod + qatest_ip
['10.1.1.1', '10.1.1.5', '10.1.1.9', '10.1.1.8', '192.168.1.5', '192.168.1.8']
>>>
```

sort()

- Default sort is in ascending order
- Returns: nothing; updates original list
- Task:
  - Sort Vlans to see which are available

```
>>> vlans = [ 300, 200, 440, 150, 450 ]
>>>
>>> vlans.sort()
>>>
>>> vlans
[150, 200, 300, 440, 450]
>>>
```

Summary

- Lists can be manipulated and values are accessed by index value
- Dictionary values are accessed by name and can be manipulated. Key-Value pairs are unordered

Data Types - Booleans

Introduction to Python for Network Engineers

Booleans

- Conditions are evaluated and evaluate to:
  - `True`
  - `False`
- Capital T and F
- No quotes!

Truth Tables

Boolean operators: and, or, not

OR	Result
True or False	True
True or True	True
False or True	True
False or False	False

AND	Result
True and False	False
True and True	True
False and True	False
False and False	False

NOT	Result
not False	True
not True	False

NOT OR	Result
not (True or False)	False
not (True or True)	False
not (False or True)	False
not (False or False)	True

NOT AND	Result
not (True and False)	True
not (True and True)	False
not (False and True)	True
not (False and False)	True

Boolean Expressions

Always evaluated to `True` or `False`

- `==`
- `!=`
- `>`
- `<`
- `in`

```
>>> hostname = 'r1'
>>> qty_cisco = 4
>>> interface_type = 'Ethernet'
>>>
>>> hostname == 'R1'
False
>>>
>>> hostname != 'R1'
True
>>>
>>> qty_cisco > 2
True
>>>
>>> 5 < qty_cisco
False
>>>
>>> 'Eth' in interface_type
True
>>>
```

Boolean Expressions (cont'd)

Using `and` and/or `or`

```
>>> hostname = 'r1'
>>> qty_cisco = 4
>>> interface_type = 'Ethernet'
>>>
>>> hostname == 'R1'
False
>>>
>>> hostname != 'R1'
True
>>>
>>> qty_cisco > 2
True
>>>
>>> 5 < qty_cisco
False
>>>
>>> 'Eth' in interface_type
True
>>>
```

```
>>> 'Eth' in interface_type and hostname != 'R1'
True
>>>
>>> hostname == 'R1' or hostname != 'R1'
True
>>>
>>> hostname == ('R1' or hostname) != 'R1'
False
>>>
>>> True and True and True or False
True
>>>
```

Precedence Rules

Highest precedence

- parentheses
- relational operators: <, >, <=, >=, !=, ==
- not
- and
- or

Lowest precedence

Lab Time

- Lab 4 - Working with Integers
  - Explore working with Integers
- Lab 5 - Working with Lists
  - Learn how to work with lists and their built-in methods while working in the Python Interactive Interpreter
- Lab 6 - Working with Booleans
  - Explore working with Booleans

Data Types - Dictionaries

Introduction to Python for Network Engineers

Dictionaries

- Dictionaries are **unordered** lists
- Instead of being indexed by a number, they are indexed by a name, more commonly known as a key
- Also known as hashes and associative arrays
- Dicts vs. Lists
    - Key Values vs. Indexed Elements

Instead of accessing an element by an index, you can use a word/string you define, i.e. a *key*

Dictionaries

- First...look at lists again...

```
>>> dev = ['sw1', '10.1.100.1/24', '00:00:00:00:00:01']
>>> # print(hostname)
>>> print(dev[0])
sw1
```

```
>>> # print(mac address)
>>> print(dev[2])
00:00:00:00:00:01
```

Accessing Values in a Dictionary

- As a list

```
>>> dev = ['sw1', '10.1.100.1/24', '00:00:00:00:00:01']
>>>
```

- Creating a Dictionary

```
>>> dev = {'hostname':'sw1', 'mgmt_ip':'10.1.100.1/24', 'mac':'00:00:00:00:00:01'}
>>>
```

Accessing Values in a Dictionary

- As a list

```
>>> dev = ['sw1', '10.1.100.1/24', '00:00:00:00:00:01']
>>>
```

- Creating a Dictionary

```
>>> dev = {'hostname':'sw1', 'mgmt_ip':'10.1.100.1/24', 'mac':'00:00:00:00:00:01'}
>>>
```

- Accessing Values

```
>>> print(dev['hostname'])
sw1
>>>
>>> print(dev['mac'])
00:00:00:00:00:01
>>>
```

Dictionaries are preferred over lists when storing contextual information and you need to programmatically access individual elements.

Creating Dictionaries

There are various options to create dictionaries

```
>>> dev1 = {'hostname': 'WAN1', 'model': '5621'}
```

Creating Dictionaries

There are various options to create dictionaries

```
>>> dev1 = {'hostname': 'WAN1', 'model': '5621'}
```

```
>>> dev2 = {}
>>> dev2['hostname'] = 'ROUTER1'
>>> dev2['model'] = 'nexus_7000'
```

Creating Dictionaries

There are various options to create dictionaries

```
>>> dev1 = {'hostname': 'WAN1', 'model': '5621'}
```

```
>>> dev2 = {}
>>> dev2['hostname'] = 'ROUTER1'
>>> dev2['model'] = 'nexus_7000'
```

```
>>> dev3 = dict(hostname='SWITCH2', model='arista_veos')
>>>
```

Creating Dictionaries

There are various options to create dictionaries

```
>>> dev1 = {'hostname': 'WAN1', 'model': '5621'}
```

```
>>> dev2 = {}
>>> dev2['hostname'] = 'ROUTER1'
>>> dev2['model'] = 'nexus_7000'
```

```
>>> dev3 = dict(hostname='SWITCH2', model='arista_veos')
>>>
```

No matter how they are created, if you need to add individual items:

```
>>> devX['vendor'] = 'cisco'
>>> devX['location'] = 'EMEA'
```

Updating a Dictionary

```
>>> location = {'city': 'nyc', 'state': 'ny', 'contact': 'jack'}
```

Overwrites existing value

```
>>> location['city'] = 'new york'
>>>
```

Add new item (key/value pair)

```
>>> location['country'] = 'usa'
>>>
```

```
>>> print(location)
{'city': 'new york', 'state': 'ny', 'contact': 'jack', 'country': 'usa'}
>>>
```

Dictionary Built-in Methods

```
>>> dir(dev)
['clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values', 'viewitems', 'viewkeys', 'viewvalues']
>>>
>>>
>>> # cleaned up for brevity
```

keys()

- Use `keys()` to see all keys in a dictionary
- Returns: `list`

```
>>> facts = {'vendor': 'cisco', 'hostname': 'NYC301', 'os': '6.1.2', 'mgmt_ip': '10.1.1.1'}
>>>
>>> facts         # notice how it is unordered and changed from above
{'os': '6.1.2', 'hostname': 'NYC301', 'vendor': 'cisco', 'mgmt_ip': '10.1.1.1'}
>>>
>>> facts.keys()
['os', 'hostname', 'vendor', 'mgmt_ip']
```

values()

- Use `values()` to see all values in a dictionary
- Returns: `list`

```
>>> facts
{'os': '6.1.2', 'hostname': 'NYC301', 'vendor': 'cisco', 'mgmt_ip': '10.1.1.1'}
>>>
>>> facts.values()
['6.1.2', 'NYC301', 'cisco', '10.1.1.1']
```

update()

- Use `update()` to add (and update) two dictionaries
- Values will be overwritten if the new value also exists in the new dictionary
- Returns: nothing; updates original dictionary

```
>>> facts
{'os': '6.1.2', 'hostname': 'NYC301', 'vendor': 'cisco', 'mgmt_ip': '10.1.1.1'}
```

```
>>> newfacts = {'model':'nexus', 'chipset':'t2'}
```

```
>>> facts.update(newfacts)
```

```
>>> facts
{'vendor': 'cisco', 'mgmt_ip': '10.1.1.1', 'chipset': 't2', 'model': 'nexus', 'hostname': 'NYC301', 'os': '6.1.2'}
```

pop()

- Remove a key-value pair from a dictionary
- Returns: value being removed and updates original dictionary

```
>>> facts
{'vendor': 'arista', 'mgmt_ip': '10.1.1.1', 'chipset': 't2', 'model': 'nexus', 'hostname': 'NYC301', 'os': '6.1.2'}
```

```
>>> facts.pop('model')
'nexus'
```

```
>>> facts
{'vendor': 'arista', 'mgmt_ip': '10.1.1.1', 'chipset': 't2', 'hostname': 'NYC301', 'os': '6.1.2'}
>>>
```

get()

- Access particular value given the key
- Ability to return user-specified value if key does not exist

```
>>> facts = {'vendor': 'arista', 'mgmt_ip': '10.1.1.1', 'chipset': 't2', 'hostname': 'NYC301', 'os': '6.1.2'}
```

When the key exists, both of these options are the same:

```
>>> facts['vendor']
'arista'
>>>
```

```
>>> facts.get('vendor')
'arista'
>>>
```

get() - (cont'd)

When the key does NOT exist:

```
>>> facts['hw_ver']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'hw_ver'
```

```
>>> facts.get('hw_ver')
>>>
```

```
>>> hw = facts.get('hw_ver')
>>> print(hw)
None
>>>
```

We will eventually be able to do `if hw:` to see if it is not `None`.

get() - (cont'd)

You can optionally return a given object (string, list, dict, etc.) if the key does not exist.

```
>>> facts.get('hw_ver', 'DNE')
'DNE'
```

```
>>> devices = {'dc-a':['rtr-a', 'rtr-b']}
>>>
>>> devices['dc-a']
['rtr-a', 'rtr-b']
>>>
>>> devices.get('dc-b', [])
[]
>>>
```

Lab Time

- Lab 7 - Working with Dictionaries
  - Learn how to work with dictionaries and their built-in methods while working in the Python Interactive Interpreter

Python Libraries

- Python modules
  - Standalone Python file used to share code between programs
- Python packages
  - Collection of Python modules

Examples:

```
import json
import os
```

json package: Pretty Printing Dictionaries

- The `json` module can be used to pretty print a dictionary
- It's serializing it as a string
- Example: Print some device facts in a much more readable indented format

```
>>> print(facts)
{'chipset': 't2', 'hostname': 'NYC301', 'vendor': 'arista', 'os': '6.1.2', 'mgmt_ip': '10.1.1.1'}
>>>
>>> import json
>>>
>>> print(json.dumps(facts, indent=4))
{
    "chipset": "t2",
    "hostname": "NYC301",
    "vendor": "arista",
    "os": "6.1.2",
    "mgmt_ip": "10.1.1.1"
}
>>>
```

os package: Operating System Tasks

- The `os` module can be used to execute operating system related tasks
- Often used to change the working directory or issue linux commands locally
- Example: Verify Google DNS is pingable from the system where the python is executed

```
>>> import os
>>>
>>> google_dns = "8.8.8.8"
>>>
>>> return_code = os.system("ping -c 1 " + google_dns)
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=59 time=11.431 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 p>>> import os
>>>
>>>
```

Example Script

Filename: `common.py`

```
#! /usr/bin/env python

command = "show run"
conf_command = "conf t ; int Gig 1 ; shutdown ;"

if __name__ == "__main__":

    print("Sending 'show' command...")
    print('Command sent: \n {}'.format(command))


    print("Sending 'config' command...")
    print('Commands sent: \n {}'.format(conf_command))
```

Example Script Output

Running `common.py` as a standalone program:

```
ntc@jump-host:~$ python common.py
Sending 'show' command...
Command sent:
 show run
Sending 'config' command...
Commands sent:
 conf t ; int Gig 1 ; shutdown ;
```

Remember, the code under the entry point conditional is only executed when the file is run as a standalone program

What if you just wanted to use variables from within `common.py` ?

Re-Usable Python Objects

What if we wanted to re-use variables from this file in another program?

Remember the filename is called `common.py`

```
#! /usr/bin/env python

command = "show run"
conf_command = "conf t ; int Gig 1 ; shutdown ;"

if __name__ == "__main__":
    # Code only executed when ran as a a program
    # More flexibility than not using the entry point when
    # you're re-suing objects in other programs
    print("Sending 'show' command...")
    print('Command sent: \n {}'.format(command))


    print("Sending 'config' command...")
    print('Commands sent: \n {}'.format(conf_command))
```

Re-Usable Python Objects

What if we wanted to re-use variables from this file in another program?

Remember the filename is called `common.py`

```
#! /usr/bin/env python

command = "show run"
conf_command = "conf t ; int Gig 1 ; shutdown ;"

if __name__ == "__main__":
    # Code only executed when ran as a a program
    # More flexibility than not using the entry point when
    # you're re-suing objects in other programs
    print("Sending 'show' command...")
    print('Command sent: \n {}'.format(command))


    print("Sending 'config' command...")
    print('Commands sent: \n {}'.format(conf_command))
```

```
>>> import common
>>>
>>> print(common.command)
show run
>>>
```

Re-Usable Python Objects

What if we wanted to re-use variables from this file in another program?

Remember the filename is called `common.py`

```
#! /usr/bin/env python

command = "show run"
conf_command = "conf t ; int Gig 1 ; shutdown ;"

if __name__ == "__main__":
    # Code only executed when ran as a a program
    # More flexibility than not using the entry point when
    # you're re-suing objects in other programs
    print("Sending 'show' command...")
    print('Command sent: \n {}'.format(command))


    print("Sending 'config' command...")
    print('Commands sent: \n {}'.format(conf_command))
```

```
>>> import common
>>>
>>> print(common.command)
show run
>>>
```

```
>>> import common
>>>
>>> print(common.conf_command)
conf t ; int Gig 1 ; shutdown ;
>>>
```

Using from/import and re-naming objects

```
>>> from common import conf_command
>>>
>>> print(conf_command)
conf t ; int Gig 1 ; shutdown ;
>>>
```

Using from/import and re-naming objects

```
>>> from common import conf_command
>>>
>>> print(conf_command)
conf t ; int Gig 1 ; shutdown ;
>>>
```

```
>>> import common
>>>
>>> print(command)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'command' is not defined
>>>
```

Using from/import and re-naming objects

```
>>> from common import conf_command
>>>
>>> print(conf_command)
conf t ; int Gig 1 ; shutdown ;
>>>
```

```
>>> import common
>>>
>>> print(command)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'command' is not defined
>>>
```

- Use `as` to rename objects as you import them
- Helpful to reduce length of long object names and eliminate naming conflicts

```
>>> from common import command as cmd
>>>
>>> print cmd
show run
>>>
```

```
>>> from common import conf_command as cfg
>>>
>>> print cfg
conf t ; int Gig 1 ; shutdown ;
>>>
```

The PYTHONPATH

- For testing, as we are doing in the course, you need to use your Python module from within the same directory where it exists
  - Enter the Python shell where the module exists
  - Write a new program and place in same directory where the module exists

OR...update your PYTHONPATH

```
ntc@jump-host:~$ env | grep "PYTHON"
PYTHONPATH=/home/ntc/python/libraries/
```

One option is to update the PYTHONPATH in `.bashrc` so changes are persistent :

```
export PYTHONPATH=$PYTHONPATH:/home/ntc/new/path
```

Summary

- JSON and OS are built-in packages which are useful
- Modules are Standalone Python files used to share code between programs
- Packages are a collection of modules

Nested Objects

Introduction to Python for Network Engineers

Nested Objects

When you're just starting, it is much more important to be able to extract data from a complex object. Common for working with device APIs.

- Basic objects include:
  - Lists of strings
  - Dictionaries with strings as values
- Need to understand more complex objects
  - One of the most important topics regardless of programming language or tool used

```
{
    "Eth1": {
        "errors": {
            "jumbo": 10,
            "crc": 5,
            "unknown": {
                "cisco": "1",
                "arista": "2"
            }
        }
    }
}
```

```
>>> print(SOME_VARIABLE['Eth1']['errors']['unknown']['cisco'])
1
>>>
```

Live Demo

- Review workflow for working with large nested objects
  - Check data type
  - Check length of lists
  - Check keys for dictionaries
  - Extract value/element
  - Repeat

Lab Time

- Lab 8 - Using Python Modules
- Lab 9 - Exploring Nested Objects
  - List of dictionaries
  - Nested dictionaries

Working with Files

Introduction to Python for Network Engineers

Sample Switch Config File

Filename: `switch.cfg`

```
hostname NYCSWITCH1

vlan 100
 name web
!
interface Ethernet 1/1
  description connecting to US101
  switchport trunk encapsulation dot1q
  switchport mode trunk

interface Ethernet 1/2
  description connecting to US102
  switchport trunk encapsulation dot1q
  switchport mode trunk

interface vlan 100
  ip address 10.100.15.100/24

ip route 0.0.0.0/0 10.100.15.1
```

Reading Data from a File

```
>>> config = open('switch.cfg', 'r')         # open file
>>>
>>> config.read()                            # read file
'hostname NYCSWITCH1\n\nvlan 100\n name web\n!\ninterface
Ethernet 1/1\n  description connecting to US101\n
switchport trunk encapsulation dot1q\n  switchport mode
trunk\n\ninterface Ethernet 1/2\n  description connecting to
US102\n  switchport trunk encapsulation dot1q\n  switchport
mode trunk\n\ninterface vlan 100\n  ip address
10.100.15.100/24\n\nip route 0.0.0.0/0 10.100.15.1'
>>>
>>> config_str = config.read()
>>>
```

Reading Data from a File

```
>>> config = open('switch.cfg', 'r')         # open file
>>>
>>> config.read()                            # read file
'hostname NYCSWITCH1\n\nvlan 100\n name web\n!\ninterface
Ethernet 1/1\n  description connecting to US101\n
switchport trunk encapsulation dot1q\n  switchport mode
trunk\n\ninterface Ethernet 1/2\n  description connecting to
US102\n  switchport trunk encapsulation dot1q\n  switchport
mode trunk\n\ninterface vlan 100\n  ip address
10.100.15.100/24\n\nip route 0.0.0.0/0 10.100.15.1'
>>>
>>> config_str = config.read()
>>>
```

```
>>> print(config_str)
hostname NYCSWITCH1

vlan 100
 name web
!
interface Ethernet 1/1
  description connecting to US101
  switchport trunk encapsulation dot1q
  switchport mode trunk

interface Ethernet 1/2
  description connecting to US102
  switchport trunk encapsulation dot1q
  switchport mode trunk

interface vlan 100
  ip address 10.100.15.100/24

ip route 0.0.0.0/0 10.100.15.1
>>>
>>> config.close()                                # close file
```

File Object & its Methods

```
>>> dir(config)
['close', 'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty', 'mode', 'name', 'newlines', 'next', 'read', 'readinto',
'readline', 'readlines', 'seek', 'softspace', 'tell', 'truncate', 'write', 'writelines', 'xreadlines']
```

Writing Data to a File

```
>>>
>>> vlans = [{'id': '10', 'name': 'USERS'}, {'id': '20',
'name': 'VOICE'}, {'id': '30', 'name': 'WLAN'}, {'id': '40',
'name': 'APP'}, {'id': '50', 'name': 'WEB'}]
>>>
>>> import json
>>> print(json.dumps(vlans, indent=4))
[
    {
        "id": "10",
        "name": "USERS"
    },
    {
        "id": "20",
        "name": "VOICE"
    },
    {
        "id": "30",
        "name": "WLAN"
    },
    {
        "id": "40",
        "name": "APP"
    },
    {
        "id": "50",
        "name": "WEB"
    }
]
>>>
```

```
>>> write_file = open("vlan_new.cfg","w")
>>>
>>> write_file.write("vlan " + vlans[0]["id"] + "\n")
>>> write_file.write("  name  " + vlans[0]["name"] + "\n")
>>>
>>> write_file.write("vlan " + vlans[1]["id"] + "\n")
>>> write_file.write("  name  " + vlans[1]["name"] + "\n")
>>>
>>> write_file.write("vlan " + vlans[2]["id"] + "\n")
>>> write_file.write("  name  " + vlans[2]["name"] + "\n")
>>>
>>> write_file.write("vlan " + vlans[3]["id"] + "\n")
>>> write_file.write("  name  " + vlans[3]["name"] + "\n")
>>>
>>> write_file.write("vlan " + vlans[4]["id"] + "\n")
>>> write_file.write("  name  " + vlans[4]["name"] + "\n")
>>>
>>> write_file.close()
>>>
```

Writing Data to a File

```
ntc@jump-host:~$ cat vlans_new.cfg
vlan 10
  name  USERS
vlan 20
  name  VOICE
vlan 30
  name  WLAN
vlan 40
  name  APP
vlan 50
  name  WEB
```

Note: always remember to close files. By default, data isn't written to the file until it's closed.

with Statement

- `with` guarantees file will be closed (context manager)

```
>>> with open('switch.cfg', 'r') as config:
...     netcfg = config.read()     # readlines() could also be used
>>> netcfg
'hostname NYCSWITCH1\n\nvlan 100\n name web\n!\ninterface Ethernet 1/1\n  description connecting to US101\n  switchport trunk
encapsulation dot1q\n  switchport mode trunk\n\ninterface Ethernet 1/2\n  description connecting to US102\n  switchport trunk
encapsulation dot1q\n  switchport mode trunk\n\ninterface vlan 100\n  ip address 10.100.15.100/24\n\nip route 0.0.0.0/0
10.100.15.1'
>>>
```

```
>>> config = open('switch.cfg', 'r')         # open file
>>>
>>> netcfg = config.read()
>>>
>>> config.close()
```

Python Scripts

Introduction to Python for Network Engineers

Executing Scripts

- Important to see how they are executed
  - Understand the user experience (always)
- `.py` file extension
- Execute with format `python scriptname.py`

```
ntc@jump-host:~$ python intro.py
Welcome to Python for Network Engineers!
This is your first script.
```

Executing Scripts

- Important to see how they are executed
  - Understand the user experience (always)
- `.py` file extension
- Execute with format `python scriptname.py`

```
ntc@jump-host:~$ python intro.py
Welcome to Python for Network Engineers!
This is your first script.
```

```
#! /usr/bin/env python

if __name__ == "__main__":

    print('Welcome to Python for Network Engineers!')
    print('This is your first script.')
```

Writing Scripts

Everything under `if __name__ == "__main__":` is the same as it would be on the Python interpreter

- `if __name__ == "__main__":` is an optional, but recommended
  - Entry point for a Python program
  - `__name__` is an internal variable set to "**main**" when the file is run as a script
- `#! /usr/bin/env python` is the shebang, optional, but recommended.
  - Tells the system which version of Python to use when running the program.

```
#! /usr/bin/env python

# filename: print_facts.py

import json

facts1 = {'vendor': 'cisco', 'os': 'nxos', 'ipaddr': '10.1.1.1'}
facts2 = {'vendor': 'cisco', 'os': 'ios', 'ipaddr': '10.2.1.1'}
facts3 = {'vendor': 'arista', 'os': 'eos', 'ipaddr': '10.1.1.2'}

devices = [facts1, facts2, facts3]

print(json.dumps(devices, indent=4))
```

Writing Scripts (cont'd)

```
#! /usr/bin/env python

# filename: print_facts.py

import json

facts1 = {'vendor': 'cisco', 'os': 'nxos', 'ipaddr': '10.1.1.1
facts2 = {'vendor': 'cisco', 'os': 'ios', 'ipaddr': '10.2.1.1'
facts3 = {'vendor': 'arista', 'os': 'eos', 'ipaddr': '10.1.1.2

devices = [facts1, facts2, facts3]

print(json.dumps(devices, indent=4))
```

```
ntc@jump-host:~$ python print_facts.py
[
    {
        "os": "nxos",
        "ipaddr": "10.1.1.1",
        "vendor": "cisco"
    },
    {
        "os": "ios",
        "ipaddr": "10.2.1.1",
        "vendor": "cisco"
    },
    {
        "os": "eos",
        "ipaddr": "10.1.1.2",
        "vendor": "arista"
    }
]
```

Scripts with Functions

- Keep the function aligned with the `if __name__ == "__main__":`

```
#! /usr/bin/env python

def get_interface_type(interface):
    if interface.lower().startswith('et'):
        itype = 'ethernet'
    elif interface.lower().startswith('vl'):
        itype = 'svi'
    elif interface.lower().startswith('po'):
        itype = 'portchannel'
    elif interface.lower().startswith('lo'):
        itype = 'loopback'
    else:
        itype = 'unknown'

    return itype

if __name__ == "__main__":

    intf = 'Ethernet2/1'
    intf_type = get_interface_type(intf)
    print(intf_type)
```

Scripts with Functions

- Keep the function aligned with the `if __name__ == "__main__":`

```
#! /usr/bin/env python

def get_interface_type(interface):
    if interface.lower().startswith('et'):
        itype = 'ethernet'
    elif interface.lower().startswith('vl'):
        itype = 'svi'
    elif interface.lower().startswith('po'):
        itype = 'portchannel'
    elif interface.lower().startswith('lo'):
        itype = 'loopback'
    else:
        itype = 'unknown'

    return itype

if __name__ == "__main__":

    intf = 'Ethernet2/1'
    intf_type = get_interface_type(intf)
    print(intf_type)
```

```
ntc@jump-host:~$ python verify_interface_type.py
ethernet
```

From Program to Shell (`-i`)

- Execute a script and get dropped into the shell when complete and you still have access to the objects within the main part of the program
- Great for testing

```
$ python -i verify_interface_type.py
ethernet
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'get_interface_type', 'intf']
>>>
>>> get_interface_type('loopback99')
'loopback'
>>>
>>> get_interface_type('portchannel5')
'portchannel'
>>>
>>> intf
'Ethernet2/1'
>>>
```

Summary

- Writing a script is no different than writing code in the Python shell
- Think about the user experience
- Continue to re-factor
  - From if/elif to loops
  - From seeing the same code a few different places to functions

Lab Time

- Lab 10 - Performing Basic File Operations
  - Read a Network Configuration File
  - Write to a Configuration File
  - Use a Context Manager
- Lab 11 - Writing Scripts
  - Hello Network Automation
  - Print Facts for Three Devices

netmiko

Python Network Libraries

Netmiko Overview

- Python library that simplifies SSH management to network devices
- Based on the Paramiko SSH library

The purposes of the 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

Supported Platforms

- Arista vEOS
- Cisco ASA
- Cisco IOS
- Cisco IOS-XR
- Cisco SG300
- HP Comware7
- HP ProCurve
- Juniper Junos
- Linux
- Brocade VDX (limited)
- Brocade ICX/FastIron (limited)
- Brocade MLX/NetIron (limited)
- Avaya ERS (limited)
- Avaya VSP (limited)
- Cisco IOS-XE (limited)
- Cisco NX-OS (limited)
- Cisco WLC (limited)
- Dell-Force10 DNOS9 (limited)
- Huawei (limited)
- Palo Alto PAN-OS (limited)
- Vyatta VyOS (limited)

Supported Platforms (experimental)

- A10
- Alcatel-Lucent SR-OS
- Enterasys
- Extreme
- F5 LTM
- Fortinet

Getting Started with Netmiko

Starting a connection to the device by creating an object that maintains the ssh session:

```
>>> from netmiko import ConnectHandler
>>>
>>> device = ConnectHandler(device_type='cisco_nxos', ip='nxos-spine1', username='ntc', password='ntc123')
>>>
```

Getting Started with Netmiko

Starting a connection to the device by creating an object that maintains the ssh session:

```
>>> from netmiko import ConnectHandler
>>>
>>> device = ConnectHandler(device_type='cisco_nxos', ip='nxos-spine1', username='ntc', password='ntc123')
>>>
```

Send a command to the device

```
>>> device.send_command('show hostname')
u'nxos-spine1 '
>>>
```

Using Netmiko

In the interactive prompt, the ssh session may time out, verify if connection is live with the `is_alive()` method which returns True/False.

```
>>> print("Connecting to CSR1...")
Connecting to CSR1...
>>> csr1 = ConnectHandler(ip='csr1', username=user, password=pwd, device_type=d_type)
>>> print("Connected to CSR1")
Connected to CSR1
>>> csr1_device_check = csr1.is_alive()
>>> print "Connected to device is verified " + str(csr1_device_check)
Connected to device is verified True
>>>
```

To reconnect if the session is lost use the `.establish_connection()` method

```
>>> csr1.disconnect()
>>> csr1.is_alive()
False
>>> csr1.establish_connection()

u''
>>> csr1.is_alive()
True
>>>
```

Using netmiko (cont'd)

Issuing Configuration changes, use send_config_set with a list of commands. Netmiko will automatically enter and exit config mode

```
>>> commands = ['interface Loopback100', 'ip address 10.200.1.20 255.255.255.0']
>>> output = csr1.send_config_set(commands)
>>> # prints commands sent
... print(output)
config term
Enter configuration commands, one per line.  End with CNTL/Z.
csr1(config)#interface Loopback100
csr1(config-if)#ip address 10.200.1.20 255.255.255.0
csr1(config-if)#end
csr1#
>>>
```

Or you can send configuration commands from a file

```
>>> csr1 = ConnectHandler(host='csr1', username='ntc', password='ntc123', device_type='cisco_ios')
>>> csr2 = ConnectHandler(host='csr2', username='ntc', password='ntc123', device_type='cisco_ios')
>>>
>>> csr1.send_config_from_file("snmp_communities.txt")
u'config term\nEnter configuration commands, one per line.  End with CNTL/Z.\ncsr1(config)#snmp-server community networktocode ro'
>>> csr2.send_config_from_file("snmp_communities.txt")
u'config term\nEnter configuration commands, one per line.  End with CNTL/Z.\ncsr2(config)#snmp-server co
>>>
```

Using netmiko (cont'd)

Storing & Printing a command response

```
>>> vlans = device.send_command('show vlans')
>>>
>>> print(vlans)

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Eth1/2, Eth1/3, Eth1/8, Eth1/9
                                                Eth1/11, Eth1/12, Eth1/13, Eth2/9
                                                Eth2/10, Eth2/11, Eth2/12
2    VLAN0002                         active    Po10, Po11, Po12, Eth1/4
                                                Eth1/5, Eth1/6, Eth1/7, Eth2/5
                                                Eth2/6
3    VLAN0003                         active    Po10, Po11, Po12, Eth1/4
                                                Eth1/5, Eth1/6, Eth1/7, Eth2/5
                                                Eth2/6
4    VLAN0004                         active    Po10, Po11, Po12, Eth1/4
                                                Eth1/5, Eth1/6, Eth1/7, Eth2/5
                                                Eth2/6
5    VLAN0005                         active    Po10, Po11, Po12, Eth1/4
                                                Eth1/5, Eth1/6, Eth1/7, Eth2/5
                                                Eth2/6

# shortened for brevity
```

Primary List of Methods

- config_mode() -- Enter into config mode
- enable() -- Enter enable mode
- establish_connection() -- Establish SSH connection to device
- exit_enable_mode() -- Exit enable mode
- find_prompt() -- Return the current router prompt
- commit() -- Execute a commit action on Juniper and IOS-XR
- disconnect() -- Close the SSH connection
- send_command_timing() - Send command down the SSH channel, return output back (uses timer to wait for device)
- send_command_expect() -- Send command to device; retrieve output until router_prompt or expect_string
- send_config_set() -- Send a set of configuration commands to remote device
- send_config_from_file() -- Send a set of configuration commands loaded from a file

Summary

- Legacy devices are here to stay (for awhile)
- Great way to bridge the gap between legacy and modern devices that return structured data
- Netmiko is a great library to integrate with TextFSM to create a pseudo-API
  - CLI commands gets sent to the device and you get returned structured data
  - We cover how to do this with Ansible

Lab Time

- Lab 12 - Exploring Netmiko
- Lab 13 - Use Netmiko to interactively communicate with a network switch
- Lab 14 - Deploying Configurations with Netmiko

Conditionals

Introduction to Python for Network Engineers

if statement

- Colon at the end of the statement is required
- 4 spaces for indentation
- Uses operators for boolean comparisons
- Task:
  - Check to see if a device is running a particular version of software. If it is, store the hostname of that device into a list.

```
>>> facts
{'vendor': 'arista', 'mgmt_ip': '10.1.1.1', 'chipset': 't2', 'hostname': 'NYC301', 'os': '6.1.2'}
```

```
>>> devices = []
>>> if facts['os'] == '6.1.2':
...     devices.append(facts['hostname'])
...
```

```
>>> print(devices)
['NYC301']
```

if Examples

Use operators such as `==` , `!=` , `>` , `<`

```
>>> hostname = 'R1'
>>>
>>> if hostname.lower() == 'r1':
...     print("Hostname Correct")
...
Hostname Correct
>>>
```

```
>>> if hostname.lower() != 'r1':
...     print("Hostname NOT Correct")
...
>>>
```

NOTE SPACING - any amount of spacing, but most important is consistency. Good practice is 4 spaces.

```
>>> if 5 > 3:
...  print("Yes, 5 is greater than 3")
...
Yes, 5 is greater than 3
>>>
```

if-else

- Colon at the end of each conditional
- 4 spaces for each conditional block
- Indentation levels must match
- **Note:**
  - `if $VAR` is `true` if the variable is not null (none/empty string), else it is `false`

```
>>> commands = ''
>>>
>>> if commands:
...     print('Commands to send: ' + commands)
... else:
...     print('No commands to send.')
...
No commands to send.
```

if-else

- Change `commands` and re-execute
- Example:
  - When writing scripts, checks should be performed to see if there are commands to send first prior to blindly sending a null string (dependent on device API)

```
>>> commands = 'config t ; interface Ethernet1/1 ; shutdown'
>>>
>>> if commands:
...     print('Commands to send: ' + commands)
... else:
...     print('No commands to send.')
...
Commands to send: config t ; interface Ethernet1/1 ; shutdown
>>>
```

if-elif-else

- Once a condition is met (true/false), the conditional block is exited
- If none are met, the else statement is executed
- Example:
  - Check to find type of a specified interface

```
>>> interface = 'Ethernet1/1'
>>>
>>> if interface.lower().startswith('et'):
...     itype = 'ethernet'
... elif interface.lower().startswith('vl'):
...     itype = 'svi'
... elif interface.lower().startswith('lo'):
...     itype = 'loopback'
... elif interface.lower().startswith('po'):
...     itype = 'portchannel'
... elif interface.lower().startswith('mgmt'):
...     itype = 'management'
... else:
...     itype = 'unknown'
>>>
>>> print(itype)
ethernet
```

if-elif-else

```
>>> interface = 'port-channel20'
>>>
>>> if interface.lower().startswith('et'):
...     itype = 'ethernet'
... elif interface.lower().startswith('vl'):
...     itype = 'svi'
... elif interface.lower().startswith('lo'):
...     itype = 'loopback'
... elif interface.lower().startswith('po'):
...     itype = 'portchannel'
... elif interface.lower().startswith('mgmt'):
...     itype = 'management'
... else:
...     itype = 'unknown'
>>>
>>> print(itype)
portchannel
```

Once the type is known, further statistics or analysis can be performed that are specific to that type of interface.

Nested Conditionals

Be very careful with indentation.

```
>>> vendor = 'cisco'
>>> platform = 'nexus'
>>> model = '9000'
>>>
>>>
>>> if vendor == "cisco":
...     if platform == "nexus":
...         if model == "9000":
...             print("Vendor:", vendor)
...             print("Platform:", platform)
...             print("Model:", model)
...         else:
...             print("unknown model")
...     else:
...         print("unknown platform")
... else:
...     print("unknown vendor")
...
Vendor: cisco
Platform: nexus
Model: 9000
>>>
```

Summary

- Conditionals are logical
- Start with basic logic:
  - if this, do that
  - if this, do that, else do something else...
- Syntax - spacing (4 spaces) and colon on each line with an `if`, `elif`, or `else`
- Remember if the variable is empty, it evaluates to False
  - `empty_list = []`
  - `empty_string = ''`
  - `empty_dict = {}`

Loops

Introduction to Python for Network Engineers

for loop

- Iterate through a given set of objects
- User-defined variable in for loop

```
>>> routers = ['r1', 'r2', 'r3']
>>>
>>> for rtr in routers:
...   print(rtr)
...
r1
r2
r3
>>>
```

Common to use `item`

```
>>> for item in routers:
...   print(item)
...
r1
r2
r3
>>>
```

for loop

Looping through a list and performing a given operation for each element in the list.

```
>>> interfaces = ['Eth1/1', 'vlan20', 'Eth4/4', 'loop10']
>>>
>>> for interface in interfaces:
...     if interface.lower().startswith('et'):
...         itype = 'ethernet'
...     elif interface.lower().startswith('vl'):
...         itype = 'svi'
...     elif interface.lower().startswith('lo'):
...         itype = 'loopback'
...     elif interface.lower().startswith('po'):
...         itype = 'portchannel'
...     elif interface.lower().startswith('mgmt'):
...         itype = 'management'
...     print(itype)
...
ethernet
svi
ethernet
loopback
>>>
```

Examples

Determine if IPs are rogue within a network when you're tracking all IP addresses and have a working list of "rogue" IP addresses.

```
>>> ips = ['1.1.1.1', '10.1.1.1', '192.168.1.1', '5.5.5.5', '10.1.2.1']
>>> rogue_ips = ['9.9.9.9', '1.1.1.1', '5.5.5.5']
>>>
>>> for ip in ips:
...     if ip in rogue_ips:
...         print('Found ROGUE IP: {}'.format(ip))
...     else:
...         print('Valid IP:', ip)
...
Found ROGUE IP: 1.1.1.1
Valid IP: 10.1.1.1
Valid IP: 192.168.1.1
Found ROGUE IP: 5.5.5.5
Valid IP: 10.1.2.1
>>>
```

Aside: Dictionary items() Method

- Commonly used with for loops to iterate over keys and values together
- `items()` simplifies accessing key/value pairs
- Returns a list of tuples and each tuple is two elements
  - Element 1 is the key and Element 2 is value
  - For now, think of tuples as a list, thus the returned object would be a list of lists

```
>>> facts
{'vendor': 'arista', 'mgmt_ip': '10.1.1.1', 'chipset': 't2', 'hostname': 'NYC301', 'os': '6.1.2'}
```

```
>>> f_list = facts.items()
>>>
>>> f_list
[('chipset', 't2'), ('hostname', 'NYC301'), ('vendor', 'arista'), ('os', '6.1.2'), ('mgmt_ip', '10.1.1.1')]
>>>
>>> len(f_list)
5
>>>
>>> f_list[0][0]
'chipset'
>>> f_list[0][1]
't2'
>>>
```

Examples

- Looping over all items in a dictionary
- Remember `dict.items()` returns a list of tuples; each tuple has two elements.
  - The first element in the tuple is the `key` and the second is the `value`.

```
>>> facts = {'chipset': 't2', 'hostname': 'NYC301', 'vendor': 'arista', 'os': '6.1.2', 'mgmt_ip': '10.1.1.1'}
>>>
>>> for key, value in facts.items():
...     print(key, value)
...
chipset t2
hostname NYC301
vendor arista
os 6.1.2
mgmt_ip 10.1.1.1
>>>
```

Examples

- Looping over all items in a dictionary
- Remember `dict.items()` returns a list of tuples; each tuple has two elements.
  - The first element in the tuple is the `fact name` (key) and the second is the `fact` itself (value).

```
>>> facts = {'chipset': 't2', 'hostname': 'NYC301', 'vendor': 'arista', 'os': '6.1.2', 'mgmt_ip': '10.1.1.1'}
>>>
>>> for device_fact, device_value in facts.items():
...     print(device_fact, device_value)
...
chipset t2
hostname NYC301
vendor arista
os 6.1.2
mgmt_ip 10.1.1.1
>>>
```

Examples

Looping over a list of dictionaries

```
>>> vlans = [{'id': '10', 'name': 'web'}, {'id': '20',
'name': 'app'}, {'id': '30', 'name': 'db'}]
>>>
>>>
>>> for vlan in vlans:
...     print('ID:', vlan['id'])
...     print('NAME:', vlan['name'])
...
ID: 10
NAME: web
ID: 20
NAME: app
ID: 30
NAME: db
>>>
```

```
>>> import json
>>> print(json.dumps(vlans, indent=4))
[
    {
        "id": "10",
        "name": "web"
    },
    {
        "id": "20",
        "name": "app"
    },
    {
        "id": "30",
        "name": "db"
    }
]
```

Summary

- The for loop is an integral part of Python, especially for Network Engineers
  - "for each element in the list, do this"
  - "for each key/value pair in the dictionary, do this"
- Syntax:
  - Colon & indentation
- When getting started...
  - Start small with a single if/elif and then re-factor to using loops

Lab Time

- Lab 15 - Getting Started with Conditionals
  - You will use conditionals to build a (simulated) list of commands that will get sent to a network device.
- Lab 16 - Getting Started with For Loops
  - You will access and print elements as you loop through lists and dictionaries
  - You will iterate through key-value pairs that are commands in the form of feature/command, and using the feature name (command key), you will access their configuration values values from another dictionary, to build a list of commands to send to a network device.
- Lab 17 Re-factoring Code Using Loops

Functions

Introduction to Python for Network Engineers

Functions

- Reusable and repeatable code should get placed into a function
- Minimize the amount of duplicate statements in your code
- Built-in function called `len`

```
>>> hostname = 'DCNJSWITCH1'
>>>
>>> len(hostname)
11
```

```
>>> commands = ['interface Ethernet1/1', 'switchport access vlan 10']
>>>
>>> len(commands)
2
>>>
```

Creating a Function

- Similar syntax to the for loop (colon and indentation)
  - Spaces, indentation, colon
- Optionally pass in arguments and return data

```
>>> def print_vlans():
...     vlans = [1, 5, 10, 11, 14, 15]
...     print(vlans)
...
>>>
```

Creating a Function

- Similar syntax to the for loop (colon and indentation)
  - Spaces, indentation, colon
- Optionally pass in arguments and return data

```
>>> def print_vlans():
...     vlans = [1, 5, 10, 11, 14, 15]
...     print(vlans)
...
>>>
```

```
>>> print_vlans()
[1, 5, 10, 11, 14, 15]
>>>
```

Returning Data from a Function

- Use the `return` statement to return an object/variable

```
>>> def get_vlans():
...     vlans = [1, 5, 10, 11, 14, 15]
...
...     return vlans
...
>>>
```

Returning Data from a Function

- Use the `return` statement to return an object/variable

```
>>> def get_vlans():
...     vlans = [1, 5, 10, 11, 14, 15]
...
...     return vlans
...
>>>
```

```
>>> vlans_list = get_vlans()
>>>
>>> print(vlans_list)
[1, 5, 10, 11, 14, 15]
>>>
```

Returning Data from a Function (cont'd)

- Pass in a variable, uses a conditional, and returns an object

```
>>> def get_vlans(device):
...     if device == 'R1':
...         vlans = [1, 5, 10, 11, 14, 15]
...     elif device == 'R2':
...         vlans = [5, 10, 14, 15]
...
...     return vlans
...
>>>
```

Returning Data from a Function (cont'd)

- Pass in a variable, uses a conditional, and returns an object

```
>>> def get_vlans(device):
...     if device == 'R1':
...         vlans = [1, 5, 10, 11, 14, 15]
...     elif device == 'R2':
...         vlans = [5, 10, 14, 15]
...
...     return vlans
...
>>>
```

```
>>> vlans = get_vlans('R1')
>>>
>>> print(vlans)
[1, 5, 10, 11, 14, 15]
>>>
```

Passing in Arguments and Returning Data

- Define parameters being passed in the function definition

```
>>> def vlan_exists(vlan_id):
...     vlans = [1, 5, 10, 11, 14, 15]
...     if vlan_id in vlans:
...         return True
...     return False
...
>>>
```

Passing in Arguments and Returning Data

- Define parameters being passed in the function definition

```
>>> def vlan_exists(vlan_id):
...     vlans = [1, 5, 10, 11, 14, 15]
...     if vlan_id in vlans:
...         return True
...     return False
...
>>>
```

```
>>> vlan_exists(10)
True
>>>
>>> if vlan_exists(25):    # you don't have to save the returned data into a var
...     print("Vlan exists")
... else:
...     print("Vlan doesn't exist")
...
Vlan doesn't exist
>>>
```

Function - Example

- Same block of code previously used
- Encapsulated in a function

```
>>> def get_interface_type(interface):
...     if interface.lower().startswith('et'):
...         itype = 'ethernet'
...     elif interface.lower().startswith('vl'):
...         itype = 'svi'
...     elif interface.lower().startswith('po'):
...         itype = 'portchannel'
...     elif interface.lower().startswith('lo'):
...         itype = 'loopback'
...     else:
...         itype = 'unknown'
...
...     return itype
...
>>>
```

Function - Example

- Same block of code previously used
- Encapsulated in a function

```
>>> def get_interface_type(interface):
...     if interface.lower().startswith('et'):
...         itype = 'ethernet'
...     elif interface.lower().startswith('vl'):
...         itype = 'svi'
...     elif interface.lower().startswith('po'):
...         itype = 'portchannel'
...     elif interface.lower().startswith('lo'):
...         itype = 'loopback'
...     else:
...         itype = 'unknown'
...
...     return itype
...
>>>
```

```
>>> print get_interface_type('Ethernet1/48')
ethernet
```

```
>>> interface_type = get_interface_type('Ethernet1/48')
>>> interface_type
'ethernet'
```

Function Calls within a for loop

- Do you really want to repeat the code in the function numerous times?
- Modularize code and separate logic

```
>>> interfaces = ['Ethernet1/1', 'loopback20', 'mgmt0']
>>>
>>> for interface in interfaces:
...     print(interface)
...     print(get_interface_type(interface))
...     print('--------')
...
Ethernet1/1
ethernet
--------
loopback20
loopback
--------
mgmt0
unknown
--------
```

Passing in Multiple Arguments

These are positional and required arguments.

```
>>> def verify_vlan(vlan_id, vlan_name):
...     print('The VLAN ID is {} while the VLAN Name is {}.'.format(vlan_id, vlan_name))
...
>>>
>>> verify_vlan(5, 'web_vlan')
The VLAN ID is 5 while the VLAN Name is web_vlan.
>>>
```

Optional Arguments

Using positional (required) arguments and optional arguments with default values.

```
>>> def interface_settings(interface, speed='auto', duplex='auto'):
...     print("Interface: ", interface)
...     print("Speed:", speed)
...     print("Duplex:", duplex)
...
>>>
```

Optional Arguments

Using positional (required) arguments and optional arguments with default values.

```
>>> def interface_settings(interface, speed='auto', duplex='auto'):
...     print("Interface: ", interface)
...     print("Speed:", speed)
...     print("Duplex:", duplex)
...
>>>
```

```
>>> interface_settings('Eth1')
Interface:  Eth1
Speed: auto
Duplex: auto
>>>
```

```
>>> interface_settings('Eth2', '1000')
Interface:  Eth2
Speed: 1000
Duplex: auto
>>>
```

Summary

- Functions provide for a means to re-use code
- From repeatedly entering the same CLI commands to writing re-usable functions
- Write a piece of code once, maybe twice, but never more!
    - Opposite of what network engineers do today

Lab Time

- Lab 18 - Getting Started with Functions
- Lab 19 - Re-factoring Code with Functions

EXTRA - BONUS

Data Types - Tuples and Sets

Introduction to Python for Network Engineers

Tuples

- Similar to lists, but elements **cannot** be added or removed
- Uses parentheses instead of brackets
- Task:
  - Create a variable that stores IP and mask as separate elements and does not permit the user to modify it.
  - System generated key-value pairs, i.e. `dict.items()`

```
>>> intf = ('192.168.1.1','24')
>>>
>>> intf
('192.168.1.1', '24')
>>>
>>> intf[0]
'192.168.1.1'
>>> intf[1]
'24'
```

Tuples

- Similar to lists, but elements **cannot** be added or removed
- Uses parentheses instead of brackets
- Task:
  - Create a variable that stores IP and mask as separate elements and does not permit the user to modify it.
  - System generated key-value pairs, i.e. `dict.items()`

```
>>> intf = ('192.168.1.1','24')
>>>
>>> intf
('192.168.1.1', '24')
>>>
>>> intf[0]
'192.168.1.1'
>>> intf[1]
'24'
```

```
>>> intf[1] = '32'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

```
>>> dir(tuple)
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__reduce_ex__', '__repr__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'count', 'index']
>>>
>>> # cleaned up for brevity
```

Data Types - Sets

Introduction to Python for Network Engineers

Sets

- Unordered list of **unique** objects
- Task:
  - Provide a list of devices that exist in the network (data comes from overlapping sources)
- Uses `set()` syntax

```
>>> device_list
['r1', 'r2', 'r5', 'r1', 'r2', 'r6', 'r8']
>>>
>>> devices_set = set(device_list)
>>>
>>> print(devices_set)
set(['r5', 'r6', 'r8', 'r1', 'r2'])
>>>
```

```
>>> device_list_updated = list(devices_set)
>>>
>>> device_list_updated
['r5', 'r6', 'r8', 'r1', 'r2']
>>>
```

Examples

```
>>> set('arista')
set(['a', 'i', 's', 'r', 't'])
>>>
```

```
>>> set(['cisco', 'cisco', 'arista', 'arista'])
set(['cisco', 'arista'])
```

Built-in Set Methods

- Display the built-in methods for a set

```
>>> dir(set)
['add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint',
'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

intersection()

- Elements that exist in two different sets
- Returns: `set`
- Task:
  - Determine which VLANs exist on two different switches

```
>>> switch_a = set(['2', '3', '10', '15', '20'])
>>>
>>> switch_b = set(['10', '15', '20', '21', '22'])
>>>
>>> switch_a.intersection(switch_b)
set(['10', '15', '20'])
```

union()

- Elements that exist in either set
- Returns: `set`
- Task:
  - Determine which VLANs that exist on either switch

```
>>> switch_a = set(['2', '3', '10', '15', '20'])
>>>
>>> switch_b = set(['10', '15', '20', '21', '22'])
>>>
>>> switch_a.union(switch_b)
set(['10', '15', '21', '22', '3', '2', '20'])
>>>
```

difference()

- See what elements one set has that another does not
- Returns: `set`
- Task:
  - View VLANs on switch A, but not on switch B

```
>>> switch_a = set(['2', '3', '10', '15', '20'])
>>>
>>> switch_b = set(['10', '15', '20', '21', '22'])
>>>
>>> switch_a.difference(switch_b)
set(['3', '2'])
>>>
```

```
>>> vlans = list(switch_a.difference(switch_b))
>>>
>>> vlans
['3', '2']
>>>
```

A quick intro to YAML

- Human readable data serialization language
- Heavily used for configuration files
- Relies heavily on indentation
- 2 space indent is common
- Superset of JSON

YAML Demo

A list of dictionaries

```yaml
---
  - vlan_name: web
    vlan_id: '10'
    vlan_state: active
  - vlan_name: app
    vlan_id: '20'
    vlan_state: active
  - vlan_name: DB
    vlan_id: '30'
    vlan_state: active
```

A nested dictionary

```yaml
---
snmp:
  ro: public
  rw: private
  info:
    location: nyc
    contact: bob
vlans:
  10:
    name: web
  20:
    name: app
```