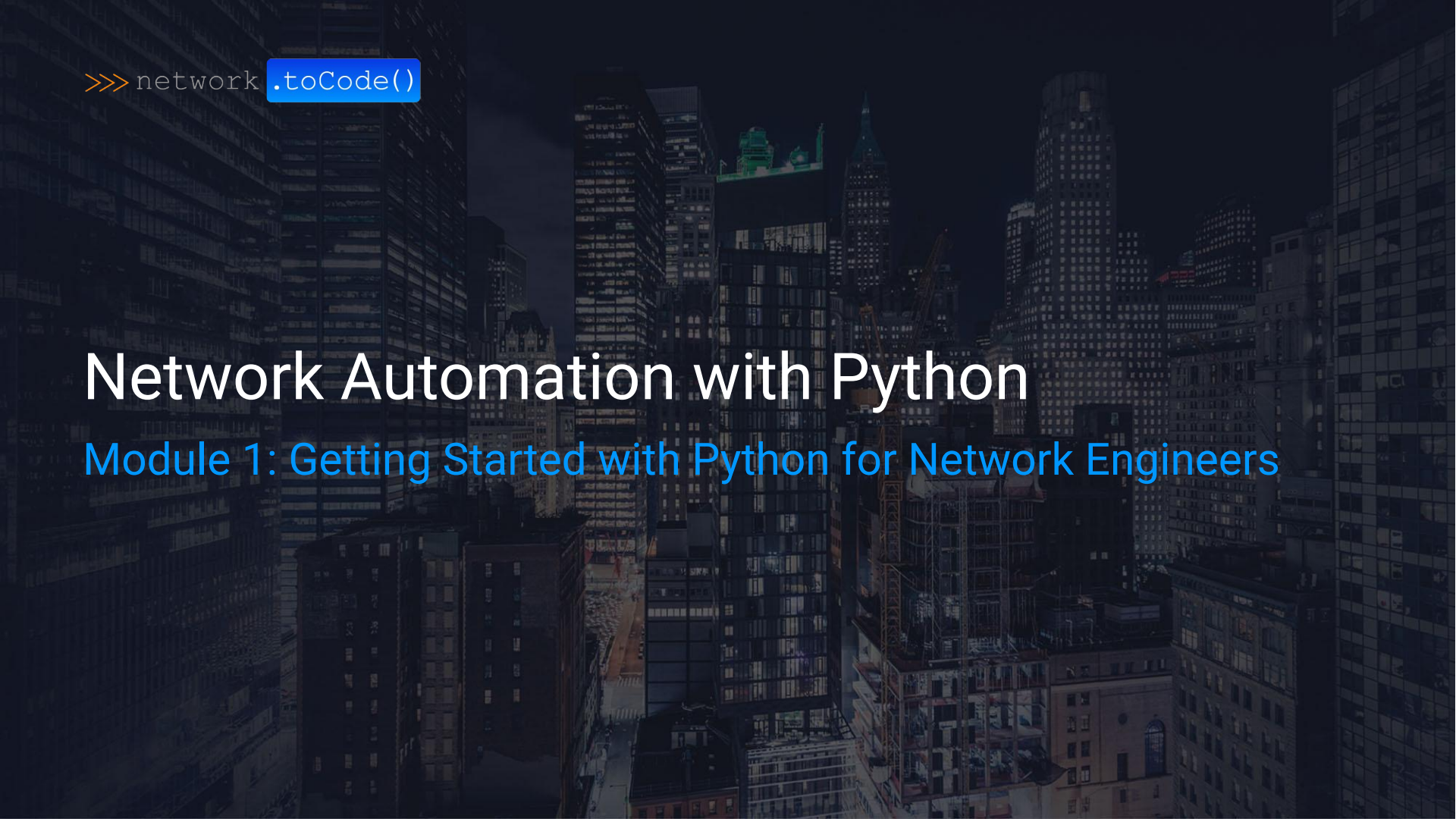
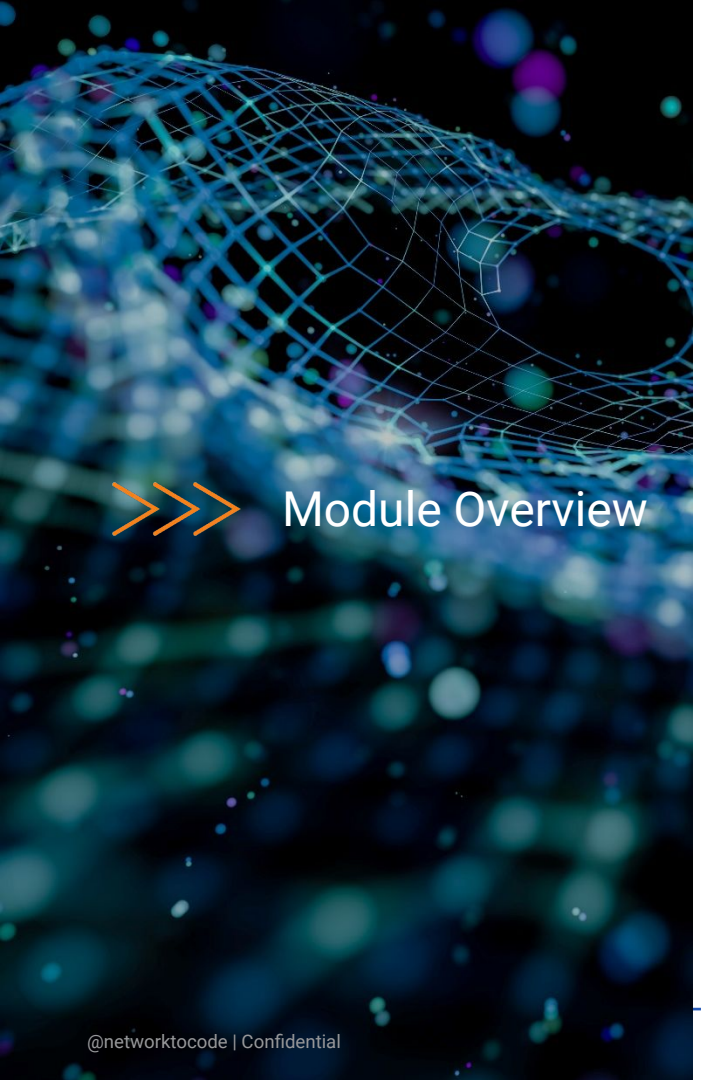

>>> network .toCode()
Network Automation with Python
Module 1: Getting Started with Python for Network Engineers



Module Overview

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

Module Table of Contents

>>> Python Interactive Interpreter

- Often called the Python Shell or REPL (Read, Eval, Print and Loop).
- 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/WSL).

```
ntc@jump-host:~$ python
Python 3.7.3 (default, Jun 29 2020, 16:17:21)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> # ENTER PYTHON CODE / STATEMENTS HERE
```

>>> Python Versions and Documentation

- The major Python version is **3.x**
  - Find the latest active versions with support windows: https://www.python.org/downloads/
  - On Linux/Mac/WSL you typically install it using the package manager of the distribution.
  - It is possible to have multiple versions of Python installed at the same time.

- Python Docs - https://docs.python.org/3/
  - Fully featured, extensive documentation is available for all Python language features and built-in functionality.
  - Make sure the version selected in the docs matches your own installation - e.g. **3.8**.x or **3.10**.x (the third number does not matter).

- There is a small chance you will encounter Python 2.7.x on older machines / environments.
  - Python 2.x is **End of Life** since early 2020!
  - You should **NOT** use it for any development unless it is unavoidable.
  - Most third party libraries you need to use for network automation have stopped supporting Python 2.

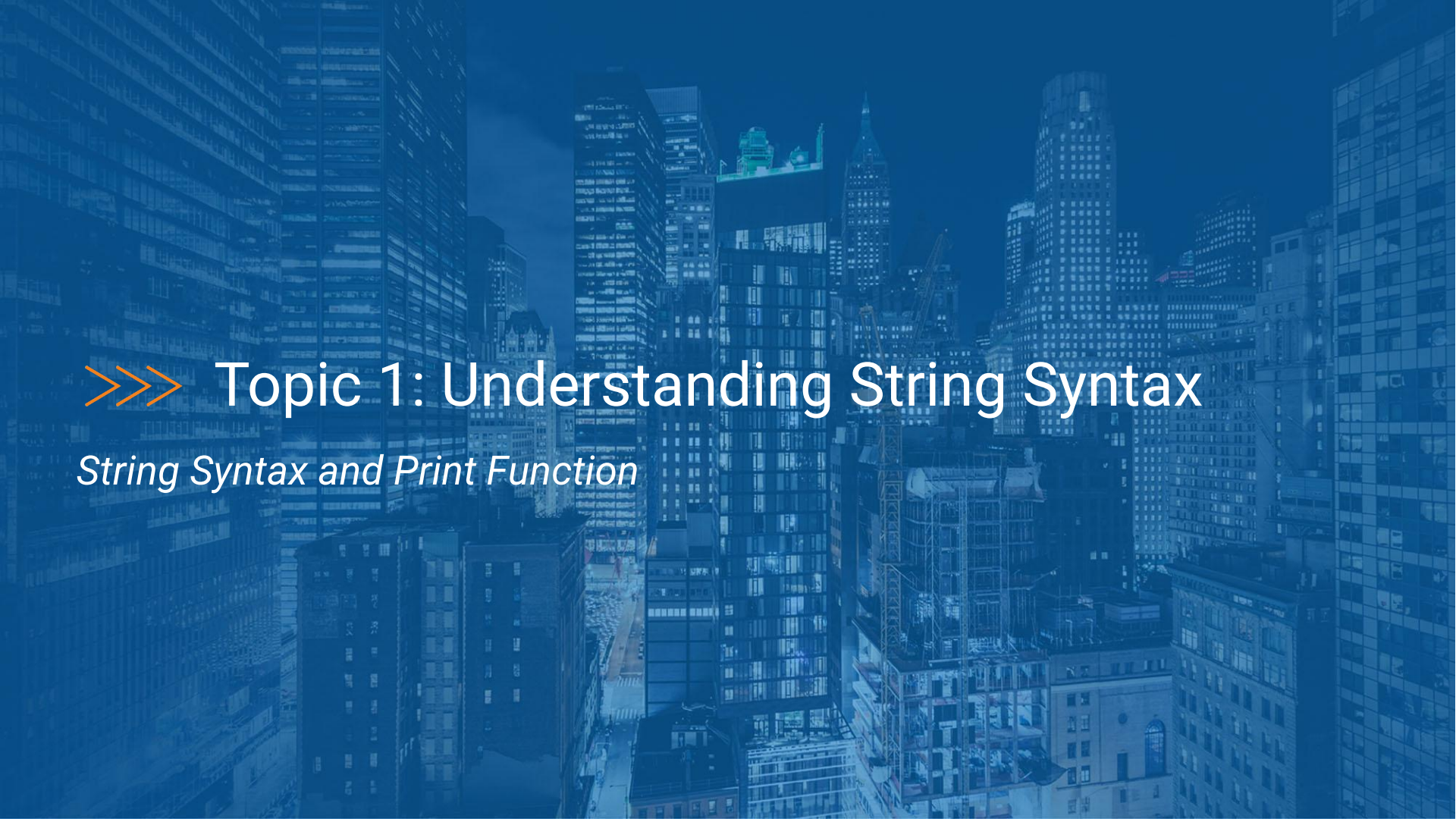
Lecture 1: Fundamental Python Data Types

Topic 1 - Understanding String Syntax

Topic 2 - Working with Python Built-In String Methods

Topic 3 - Python White Space and Style Guide

Labs 1-3



Topic 1: Understanding String Syntax

String Syntax and Print Function

>>> 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"

```
Python 3.7.3 (default, Jun 29 2020, 16:17:21)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>> hostname = "ROUTER1"
```

>>> Strings

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

```
Python 3.7.3 (default, Jun 29 2020, 16:17:21)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>> hostname = "ROUTER1"
>>>
>>> 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)
>>> <class 'str'>
>>>
```

str denotes the object is a String

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

>>> Printing Strings

With the print statement:

```
>>> ip_addr = "10.100.1.1"
>>>
>>> print(ip_addr)
10.100.1.1
>>>
```

```
>>> ip_addr
>>>
'10.100.1.1'
>>>
```

>>> Printing Strings

- 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, creating a new one in the process.

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

>>> 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...

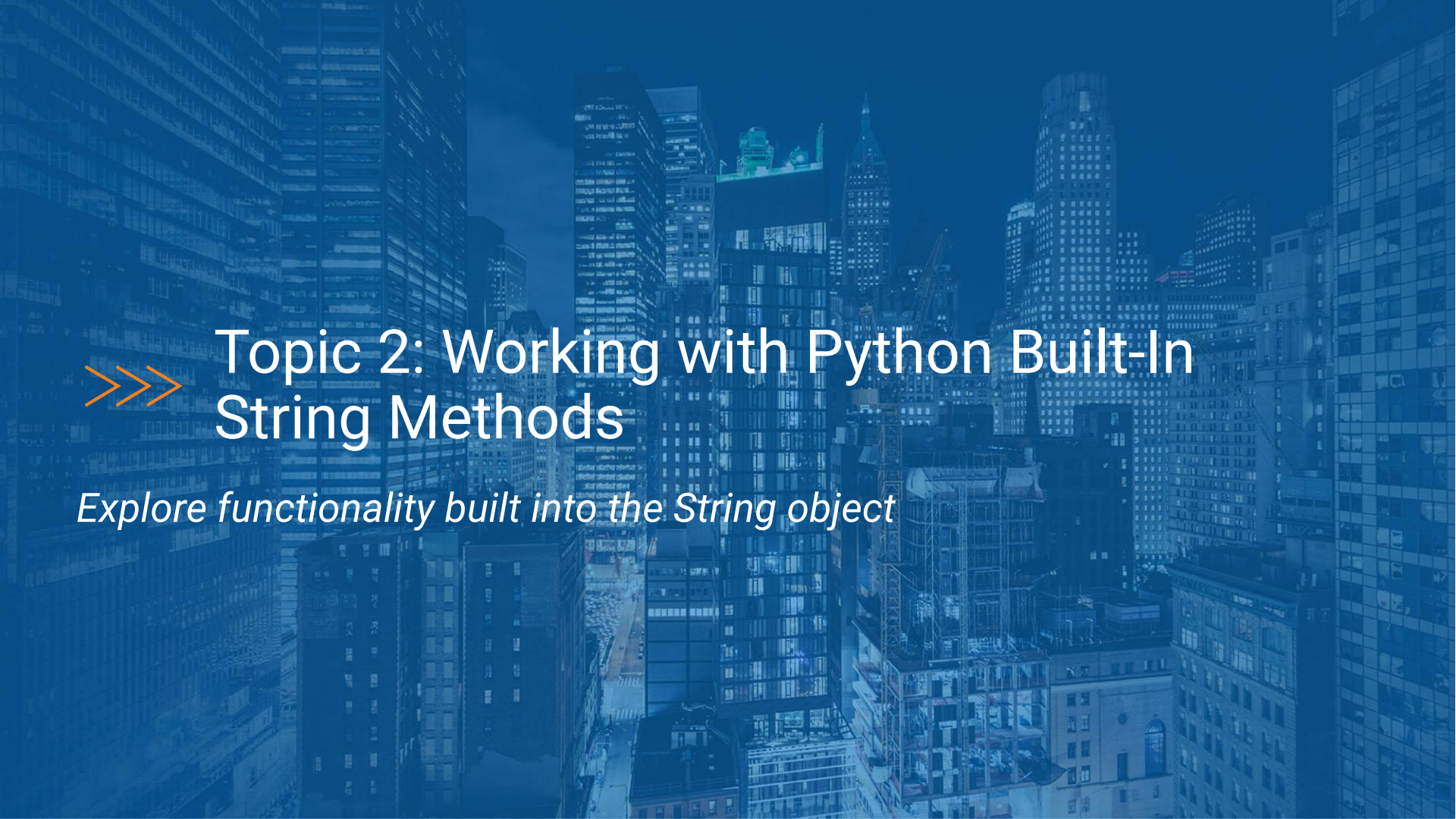
```
>>> print("The IP Address and Mask is: " + ipmask)
The IP Address and Mask is: 10.1.100.1/24
>>>
>>> statement = "The IP Address and Mask is: " + ipmask
>>> statement
'The IP Address and Mask is: 10.1.100.1/24'
>>>
```

>>> Printing Repeating Strings

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

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

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



Topic 2: Working with Python Built-In String Methods

Explore functionality built into the String object

>>> 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)
<class '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)
>>>
```

>>> The “in” membership operator

- Allows you to check if a **string** is **contained** in another **string**.
- It works with other data types that function as containers
  - Remember: a string is a sequence of characters!
- It does not tell you ***where*** it is located or whether there are multiple matches - just a simple yes(**True**) or no(**False**).

```
>>> ipmask
'10.1.100.1/24'
>>> "10" in ipmask
True
>>> "20" in ipmask
False
```

>>> 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)
```

>>> Stringing string methods together

- You can string together multiple string methods one after the other to apply more operations in the same statement (or code line).
- The resulting expression is evaluated left-to-right, one method call at a time.
- Each method call assumes that the result of the previous one is a string in this case.

```
>>> hostname = 'NYCswitch01'
>>> hostname.upper()
'NYCSWITCH01'
>>> hostname.lower()
'nycswitch01'
>>> hostname.upper().lower().startswith('nyc')
True
```

>>> 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 whitespace

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

Remove leading whitespace

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

Remove trailing whitespace

```
>>> 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)
<class '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)
<class '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 mainly with lists, but showing for completeness

>>> 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
>>> 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
>>>
```

>>> Print formatting with f-strings

- Formatted string literals or **“f-strings”** are generally considered more readable, more concise, and less prone to error than earlier approaches.
- They are string literals prefixed with the letter **f** (see below for an example).
- Available since Python **3.6**.

```
>>> hostname = 'nxos-spine1'
>>> vendor = 'cisco'
>>>
>>> print('Device hostname is {}, shipped from {}'.format(hostname, vendor))
Device hostname is nxos-spine1, shipped from cisco
>>>
>>> # same result with f-strings, with clean and compact syntax
>>>
>>> print(f'Device hostname is {hostname}, shipped from {vendor}')
Device hostname is nxos-spine1, shipped from cisco
>>>
```

Topic 3: Python White Space and Style Guide

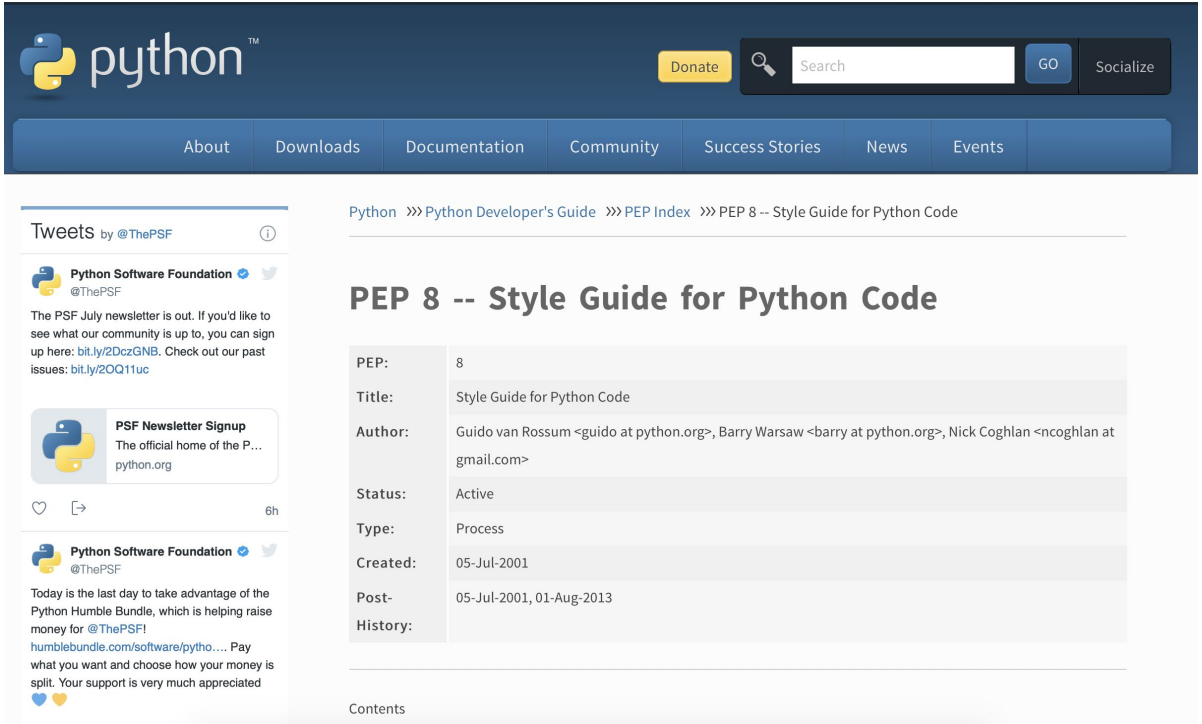
A few tips on writing good Python code

>>> White Space

- Generally the amount of space is not that important
- Just be consistent
- Tabs vs. Spaces
- PEP8 - https://www.python.org/dev/peps/pep-0008/
- The examples below work, but are not good habits to form

```
>>> # This is correct
>>> hostname = 'router1'
>>>
>>> # These work, but are not good practice
>>> hostname='router1'
>>>
>>> hostname       =            'router1'
>>>
```

>>> Style - PEP8


python
Donate
Search
GO
Socialize
About
Downloads
Documentation
Community
Success Stories
News
Events
Tweets by @ThePSF
Python Software Foundation
@ThePSF
The PSF July newsletter is out. If you'd like to see what our community is up to, you can sign up here: bit.ly/2DczGNB. Check out our past issues: bit.ly/2OQ11uc
PSF Newsletter Signup
The official home of the P…
python.org
6h
Python Software Foundation
@ThePSF
Today is the last day to take advantage of the Python Humble Bundle, which is helping raise money for @ThePSF! humblebundle.com/software/pytho…. Pay what you want and choose how your money is split. Your support is very much appreciated
Python >>> Python Developer's Guide >>> PEP Index >>> PEP 8 -- Style Guide for Python Code
PEP 8 -- Style Guide for Python Code
PEP: 8
Title: Style Guide for Python Code
Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status: Active
Type: Process
Created: 05-Jul-2001
Post-History: 05-Jul-2001, 01-Aug-2013
Contents

>>> 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!
```

>>> Summary

- Use the Python Interpreter… often.
- type(), dir(), help() are your best friends.
- 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?

Lecture 1: Fundamental Python Data Types

Labs 1-3

>>> 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 Guides: Use the README.md file in the root of the GitHub repository.

Lecture 2: Fundamental Python Data Types

Topic 4 - Numbers

Topic 5 - Lists

Topic 6 - Booleans

Labs 4-6

Topic 4: *Numbers*

Understanding Integer and Float Data Types

>>> 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
>>>
```

Topic 5: Lists

Building a List, Accessing a List

Built-in List Methods

>>> 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']
```

```
>>> 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'
>>>
>>> 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]
>>>
```

>>> The “in” membership operator

- Allows you to check if an **object** is contained in a **list**.
  - Lists can contain any Python object, so the type does not have to match like in the case of strings.
- It does not tell you ***where*** it is located or whether there are multiple matches - just a simple yes(**True**) or no(**False**).

```
>>> vlans
[300, 200, 440, 150, 450]
>>> 200 in vlans
True
>>> "200" in vlans
False
>>>
```

>>> Summary

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

Topic 6: Booleans

Truth Tables

Booleans Expressions

Precedence Rules

>>> 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)

Examples:

```
>>> 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: `()`, `[]`, `{}`
- comparisons: `<`, `>`, `<=`, `>=`, `!=`, `==`, `in`
- not
- and
- or

Lowest precedence

Full list for reference: https://docs.python.org/3/reference/expressions.html#operator-precedence

Lecture 2: Fundamental Python Data Types

Labs 4-6

>>> 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

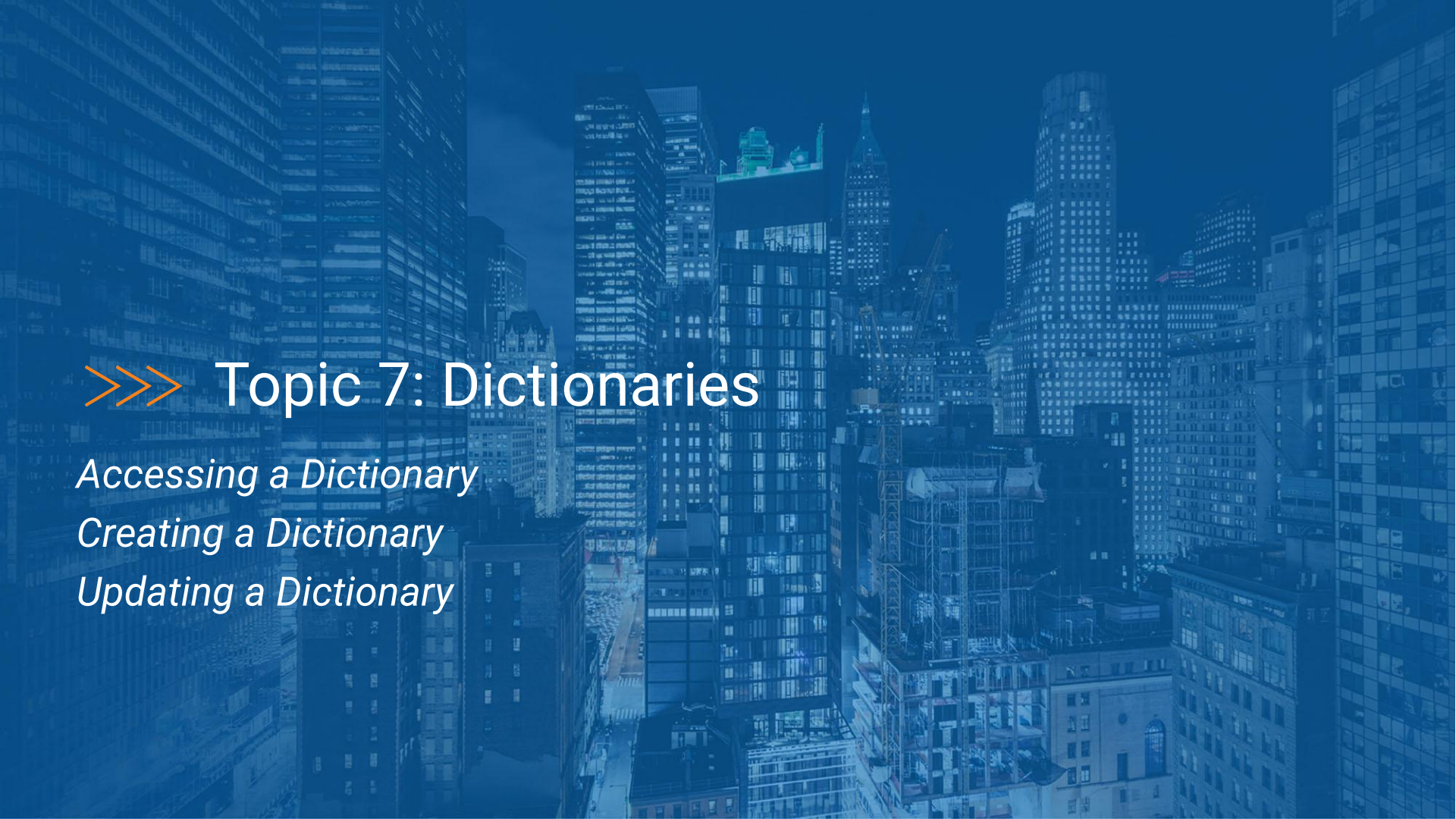
Lecture 3: Working with Dictionaries and Nested Objects

Topic 7 - Dictionaries

Topic 8 - Python Libraries

Topic 9 - Nested Objects

Labs 7-9



Topic 7: Dictionaries

Accessing a Dictionary

Creating a Dictionary

Updating a Dictionary

>>> Dictionaries

- Dictionaries are unordered lists - or rather order does not matter (in most cases)
  - From Python 3.7, dictionaries are guaranteed to keep insertion order for the keys as they are added
- 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… let's 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
```

>>> Creating 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'}
>>>
```

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

>>> 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
>>>
```

>>> 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

Overwrites existing value

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

```
>>> 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: a list-like live view

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

>>> values()

- Use values() to see all values in a dictionary
- Returns: a list-like live view

```
>>> facts
{'os': '6.1.2', 'hostname': 'NYC301', 'vendor': 'cisco', 'mgmt_ip': '10.1.1.1'}
>>>
>>> facts.values()
dict_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 use an **if** statement to check 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', [])
[]
>>>
```

>>> Bonus: Dictionary IDENTICAL vs EQUAL

- **dict2** is both EQUAL TO (==) & IDENTICAL TO (is) **dict1**

```
>>> dict1 = { 'key1': 'value1', 'key2': 'value2', 'key3': 'value3' }
>>> dict1
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
>>> dict2 = dict1
>>> dict2
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
>>> dict2 == dict1
True
>>> dict2 is dict1
True
>>>
```

>>> Bonus: Dictionary IDENTICAL vs EQUAL

- **dict2**'s key1 value ALSO changed because **dict2** is IDENTICAL TO **dict1**

```
>>> dict1
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
>>> dict2
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
>>>
>>> dict1['key1'] = 'new value'
>>>
>>> dict1
{'key1': 'new value', 'key2': 'value2', 'key3': 'value3'}
>>>
>>> dict2
{'key1': 'new value', 'key2': 'value2', 'key3': 'value3'}
>>>
```

>>> Bonus: Dictionary IDENTICAL vs EQUAL

- **dict3** has the same values as **dict2** (logically EQUAL)
- but **dict3** and **dict2** are NOT IDENTICAL (`'is'` returns FALSE)

```
>>> dict3 = dict(dict2)
>>> dict3
{'key1': 'new value', 'key2': 'value2', 'key3': 'value3'}
>>> dict2
{'key1': 'new value', 'key2': 'value2', 'key3': 'value3'}
>>> dict3 == dict2
True
>>> dict3 is dict2
False
```

>>> Bonus: Dictionary IDENTICAL vs EQUAL

- **dict3** and **dict2** are NOT IDENTICAL (`'is'` returns FALSE)
- So changing a dictionary value for **dict3** does NOT affect others (e.g. **dict2** or **dict1**)

```
>>> dict3['key3'] = 'another new value'
>>> dict3
{'key1': 'new value', 'key2': 'value2', 'key3': 'another new value'}
>>> dict2
{'key1': 'new value', 'key2': 'value2', 'key3': 'value3'}
>>> dict1
{'key1': 'new value', 'key2': 'value2', 'key3': 'value3'}
>>>
>>> dict3 == dict2
False
>>> dict3 is dict2
False
>>> dict3 == dict1
False
>>> dict3 is dict1
False
```

>>> Bonus: Dictionary IDENTICAL vs EQUAL

- **dict3** and **dict2** are NOT IDENTICAL (`'is'` returns FALSE)
- But EQUALITY can be re-established with the same key/value pairs

```
>>> dict3['key3'] = 'value3'
>>>
>>> dict3
{'key1': 'new value', 'key2': 'value2', 'key3': 'value3'}
>>> dict2
{'key1': 'new value', 'key2': 'value2', 'key3': 'value3'}
>>>
>>> dict3 == dict2
True
```

Topic 8: Python Libraries

JSON Package

OS Package

>>> 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 received, 0% packet loss, time 0ms
>>>
>>>
```

>>> 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

Topic 9: Nested Objects

Review Workflow for Nested Objects

>>> 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

Lecture 3: Working with Dictionaries and Nested Objects

Labs 7-9

>>> Lab Time

- Lab 7 - Working with Dictionaries
- Lab 8 - Using Python Modules
- Lab 9 - Exploring Nested Objects
  - List of dictionaries
  - Nested dictionaries

Lecture 4: Working with Files in Python Scripts

Topic 10 - File Operations

Topic 11 - Python Scripts

Labs 10-11

Topic 10: File Operations

Reading Data from a File

Writing Data to a File

>>> 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()
>>>
```

```
>>> 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()
```

Topic 11: Python Scripts

Executing Python code saved in a file on disk

>>> 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

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

>>> 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"
    }
]
```

>>> 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

if __name__ == "__main__":

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

>>> 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 interpreter 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'
>>>
```

>>> 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(f'Command sent: \n {command}')


    print("Sending 'config' command...")
    print(f'Commands sent: \n {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(f'Command sent: \n {command}')


    print("Sending 'config' command...")
    print(f'Commands sent: \n {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

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

```
>>> 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
>>>
```

```
>>> 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

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

- OR… update your PYTHONPATH
  - One option is to update the PYTHONPATH in .bashrc so changes are persistent:

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

>>> Summary

- Writing a script is no different than writing code in the Python shell
- Think about the user experience

Lecture 4: Working with Files in Python Scripts

Labs 10-11

>>> 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