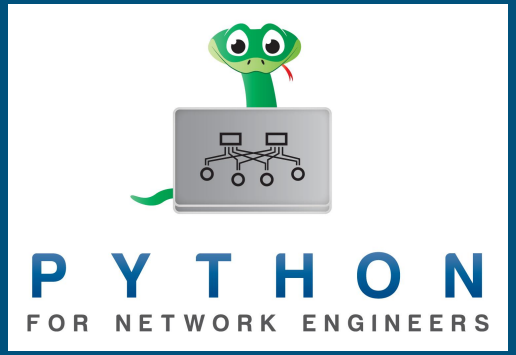
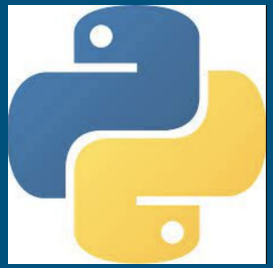




Virtual Training Session
November 2022 (Day3)

Day3 Presentation:

https://github.com/twin-bridges/pynet-ons-oct22/blob/main/python-training-day3.pdf

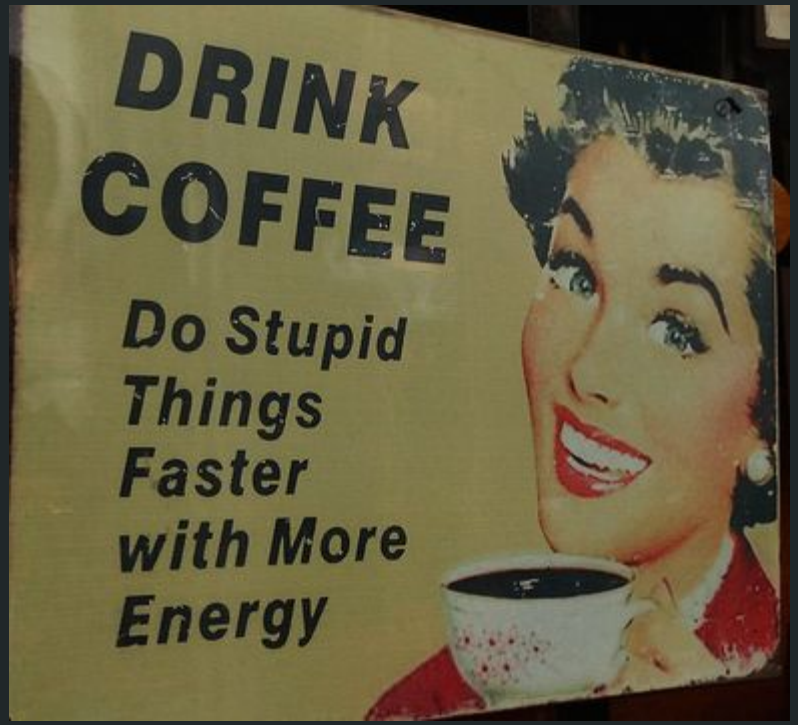
General:

Virtual-Training Day
Nov 4th, Day3 (Fri) / 9AM - 4:30PM Central

Two-Day Live Training
Nov 15th - 16th (Tues/Weds)

Focused/Minimize Distractions

The exercises are important.



Flickr: Ben Sutherland

Function Syntax

'def' keyword indicates function definition

Indented code block

Function name

```
def my_func(arg1, arg2, arg3=None):
    print("This is a function")
    print(f"arg1 value --> {arg1}")

    return arg1 + arg2

# Call the function
my_func(22, 33)
```

Return statement (implicit return None)

Function Syntax

```
def my_func(arg1, arg2, arg3=None):
    print("This is a function")
    print(f"arg1 value --> {arg1}")

    return arg1 + arg2

# Call the function
my_func(22, 33)
```

You have to call the function or nothing happens

Functions - The Why?

Conjunction junction, what's your function.

Write something once, use multiple times

```
def my_func(arg1, arg2, arg3=None):
    print("This is a function")
    print(f"arg1 value --> {arg1}")

    return arg1 + arg2

# Call the function
my_func(22, 33)
```

Functions - Exercise1

1. Create a function named “print_hello”.
2. It takes no arguments and prints the message “hello world” three times.
3. Have your program call this function three times in a row.
4. You should see “hello world” printed nine times in total.

GitHub: {{ repo }}/day3/functions/exercise1.txt

Function Parameters and Arguments

Function Parameter (this variable only exists in the function)

```
def display_output(output):
    print()
    print("#" * 80)
    print("CFG Change: ")
    print(output)
    print("#" * 80)
    print()

display_output("Whatever")
```

Function Argument - what you pass-in on the function call.

Functions - Exercise2

1. Expand on exercise 1 except the message you print out is NOT “hello world” instead it is a parameter named “msg” that is defined in the function definition.
2. Call your function, three different times and pass in three different arguments (i.e. pass in three different messages).

GitHub: {{ repo }}/day3/functions/exercise2.txt

Functions: More than one parameter.

Positional arguments: first-to-first; second-to-second, etc

```
def display_output(msg1, msg2):
    print()
    print("#" * 80)
    print(f"msg1: {msg1}")
    print("-" * 80)
    print(f"msg2: {msg2}")
    print("#" * 80)
    print()

display_output("Message1", "Message2")
display_output("Hello", "Something")
```

Functions: Using named arguments

Explicitly tell Python via naming the arguments.

```
def display_output(msg1, msg2):
    print()
    print("#" * 80)
    print(f"msg1: {msg1}")
    print("-" * 80)
    print(f"msg2: {msg2}")
    print("#" * 80)
    print()

display_output(msg2="Hello", msg1="Something")
```

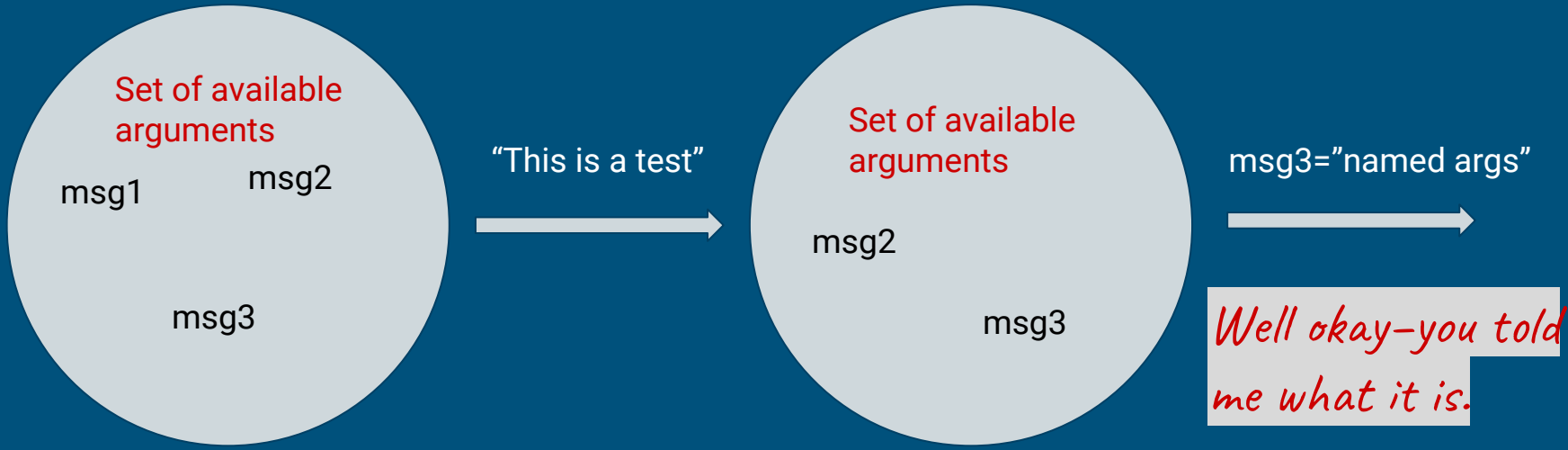
Functions: Mixing and matching positional arguments with named args

```
def display_output(msg1, msg2, msg3):
    print(f"msg1: {msg1}")
    print(f"msg2: {msg2}")
    print(f"msg3: {msg3}")


display_output("This is a test", msg3="named args", msg2="of positional and")
```

Positional arguments first

Named arguments last

```python
def display_output(msg1, msg2, msg3):
    print(f"msg1: {msg1}")
    print(f"msg2: {msg2}")
    print(f"msg3: {msg3}")


display_output("This is a test", msg3="named args", msg2="of positional and")
```



Functions - Exercise3

1. Create a function that has four parameters named var1, var2, var3, var4.
2. In the function print out each variable and indicate which variable it is.
3. Call the function using entirely positional arguments.
4. Call the function using entirely named arguments.
5. Call the function with var1 as a positional argument and var2 through var4 as named arguments.
6. Try to call the function with var1 specified first and using a named argument (and var2 through var4 as positional arguments, but specified after var1). This will generate an error.

GitHub: {{ repo }}/day3/functions/exercise3.txt

Functions: Default Values

msg3 parameter has a default value

```
def display_output(msg1, msg2, msg3="Hello World"):
    print()
    print("#" * 80)
    print(f"msg1: {msg1}")
    print(f"msg2: {msg2}")
    print(f"msg3: {msg3}")
    print("#" * 80)
    print()


# Note, msg3 argument is NOT specified here
display_output(msg2="Hello", msg1="Something")
```

Note: Very useful for expanding functions across time.

Functions: ReturnValues

```
def test_func(x, y, z):
    return x + y + z


result = test_func(7, 9, 1)
print(result)
```

Functions can return results which can then be used outside of the function.

Functions: Where do functions look for variables (LEGB Rule)

LEGB:
L = Local
E = Enclosed (nested functions)
G = Global
B = Builtins

```
IP_ADDR = "1.1.1.1"

def display_output(msg1):
    print()
    msg2 = "Locally defined variable"
    print("#" * 80)
    print(f"msg1: {msg1}")
    print(f"msg2: {msg2}")
    # Print out a global variable
    print(f"IP Addr: {IP_ADDR}")
    print("#" * 80)
    print()
```

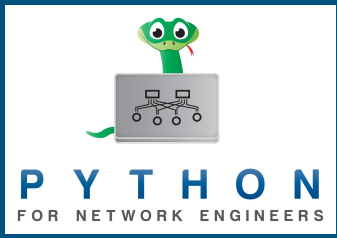
msg1 = Local variable

msg2 = Local variable

IP_ADDR = Global variable

print() = Builtin

Functions: Things that might not be obvious.



Function parameters/arguments don't have to be strings. They can be other data types including numbers and potentially lists & dictionaries.

Functions always return something (there is a default return None).

Do not directly use lists or dictionaries with default values. Instead do:

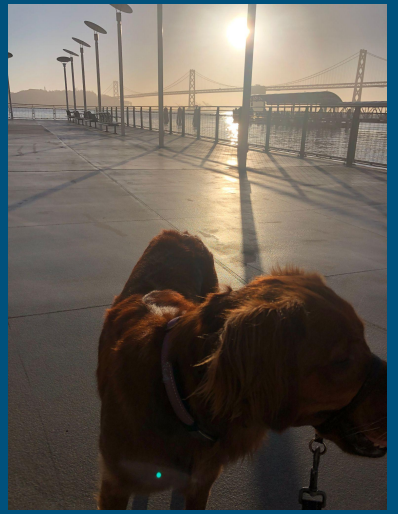
```
# Do NOT do this!
def display_output(var1, var2, var3=[]):
    print("Hello")
```

```
# Instead, do this
def display_output(var1, var2, var3=None):
    if var3 is None:
        var3 = []
    print("Hello")
```

Functions: Additional topics you can explore.

1. Passing arguments using *args.
2. Passing arguments using **kwargs.
3. Defining parameters using **kwargs.
4. Python's Lambda function.

Functions - Exercise4



Based on your earlier exercise where you parsed the serial number from “show version” output (or use ex4_reference_base.py)

a. Create two functions

b. Function1 opens the file and returns all of the data in the file as a text string. This function should take one argument (the filename).

c. Function2 parses the show_version output and returns the serial number.

GitHub: {{ repo }}/day3/functions/exercise4.txt

GitHub: {{ repo }}/day3/functions/ex4_reference_base.py

Exception Handling

Why - Gracefully handle errors.

```
In [2]: my_dict["no_key"]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-2-56d0da37b330> in <module>
----> 1 my_dict["no_key"]

KeyError: 'no_key'
```

```
In [3]: my_list = []

In [4]: my_list[7]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-4-352a83797fff> in <module>
----> 1 my_list[7]

IndexError: list index out of range
```

Exception Handling

Why - Gracefully handle errors.

```
In [5]: for i in range(10):
   ...:     print(i)
   ...:      print("Indentation off")
  File "<ipython-input-5-ceba4aa1ecbd>", line 3
    print("Indentation off")
    ^
IndentationError: unexpected indent
```

```
In [6]: forx i in range(10):
   ...:     print(i)
  File "<ipython-input-6-dec1f37c6ba2>", line 1
    forx i in range(10):
         ^
SyntaxError: invalid syntax
```

Exception Handling

Why - Gracefully handle errors.

```
In [7]: "hello" + 12
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-e8992ec33927> in <module>
----> 1 "hello" + 12

TypeError: can only concatenate str (not "int") to str
```

```
In [8]: open("bogus_file.txt")
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-8-a6a0a3b54d04> in <module>
----> 1 open("bogus_file.txt")

FileNotFoundError: [Errno 2] No such file or directory: 'bogus_file.txt'
```

Exception Handling

Actually handling the exception - this particular error might happen here.

```
my_ds = {}
try:
    # An error might happen here
    my_ds["invalid_key"]
except KeyError:
    # The specified error happened - what do I do about it.
    print("Handling KeyError exception")
```

Exception Handling

Actually handling the exception - this particular error might happen here.

```
my_list = []
try:
    # Another error might happen
    my_list[7]
except IndexError:
    # The error happened - handle it
    print("The given list index didn't exit")
```

Handling Generic Exceptions

Be careful - you could be hiding errors!

```
my_ds = {}
my_list = []
try:
    # An error might happen here
    my_ds["invalid_key"]
    my_list[7]
except Exception:
    # An error happened - keep going.
    print("Generic exception handling")
```

Finally: Do Something in Either Case

```
my_ds = {}
my_list = []
try:
    # An error might happen here
    my_ds["invalid_key"]
    my_list[7]
except Exception:
    # An error happened - keep going.
    print("Generic exception handling")
finally:
    print("Error or no error - print this message out")
```

You can "raise" your own exceptions.

```python
my_ip = "192.168.1.1"
if my_ip != "10.1.1.1":
    raise ValueError(f"You are connecting to the wrong device:\n\n{my_ip}")
```

```
(.venv) [ktbyers@pydev2 EP]$ python invalid_ip.py
Traceback (most recent call last):
  File "/home/ktbyers/EP/invalid_ip.py", line 3, in <module>
    raise ValueError(f"You are connecting to the wrong device:\n\n{my_ip}")
ValueError: You are connecting to the wrong device:

192.168.1.1
```

Exceptions - Exercise1

Expand on this exercise.

Based on your earlier exercise where you parsed the serial number from “show version” output

a. Create two functions

b. Function1 opens the file and returns all of the data in the file as a text string. This function should take one argument (the filename). If the given file is not found gracefully handle the error and print a message to the screen (return an empty string from the function).

c. Function2 parses the show_version output and returns the serial number.

GitHub: {{ repo }}/day3/exceptions/exercise1.txt

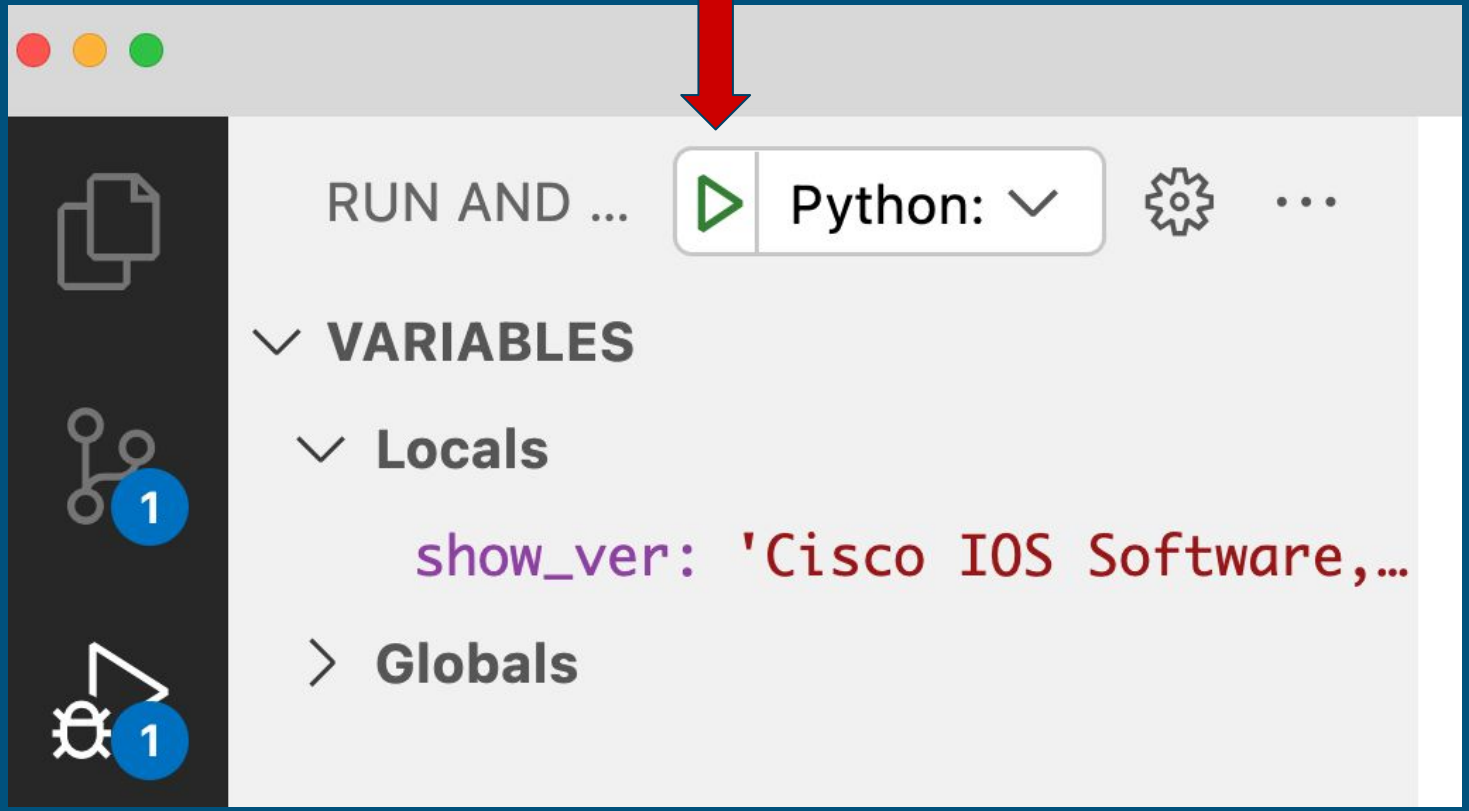
VS Code Debugging

```
5
6   def find_serial_number(show_ver):
7       serial_number = ""
8       for line in show_ver.splitlines():
9           if "Processor board ID" in line:
```

Setting a breakpoint

Executing the Debugger

Run

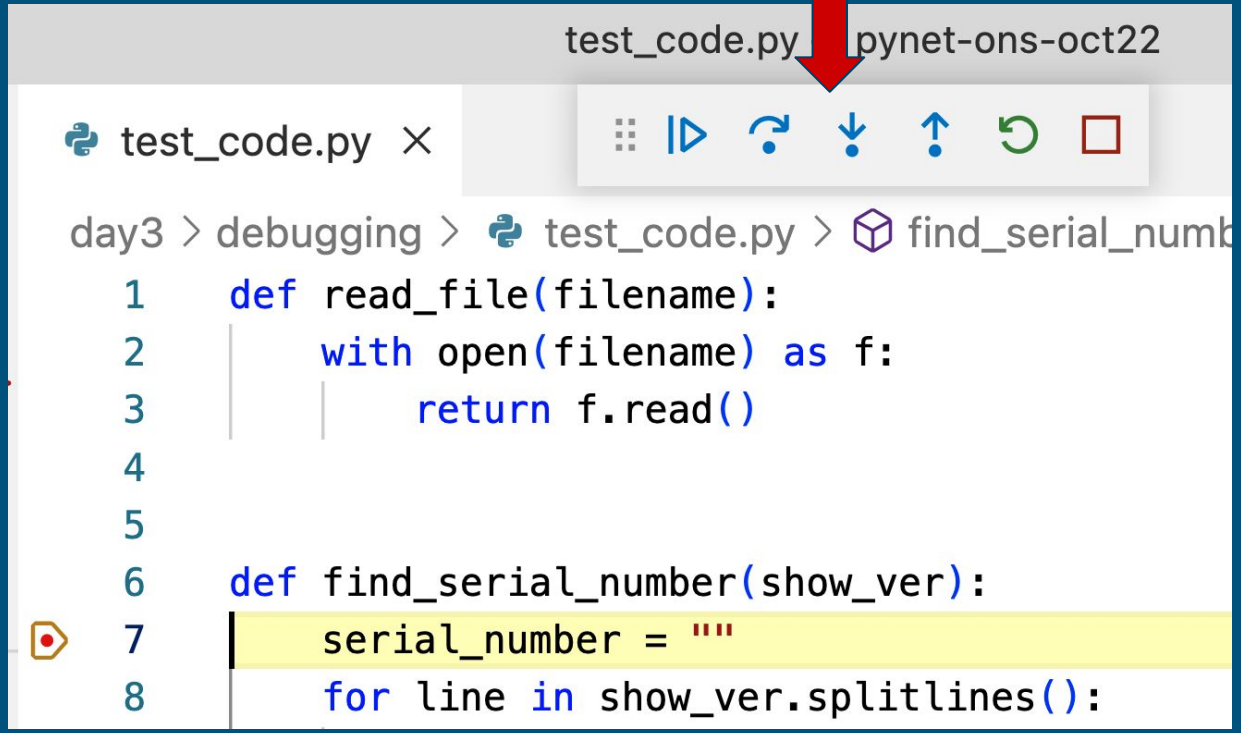


Run & Debug

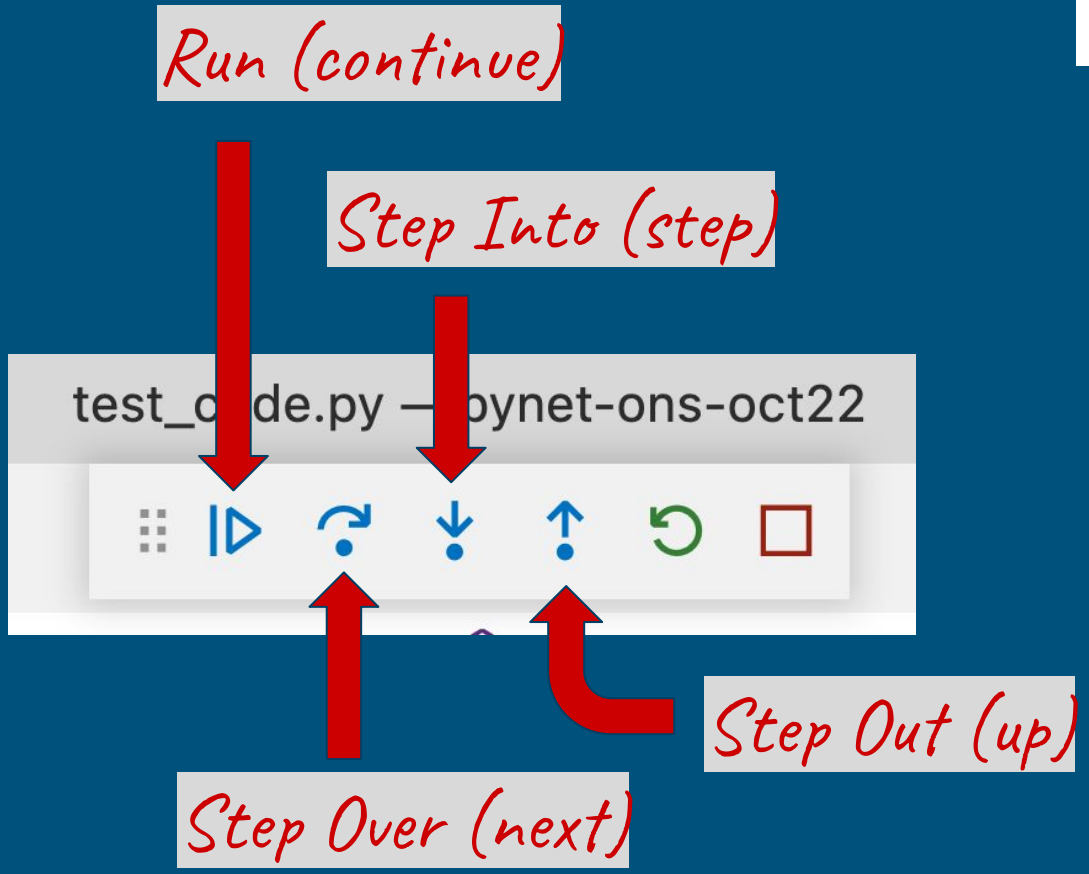
Executing the Debugger

Debugger Toolbar

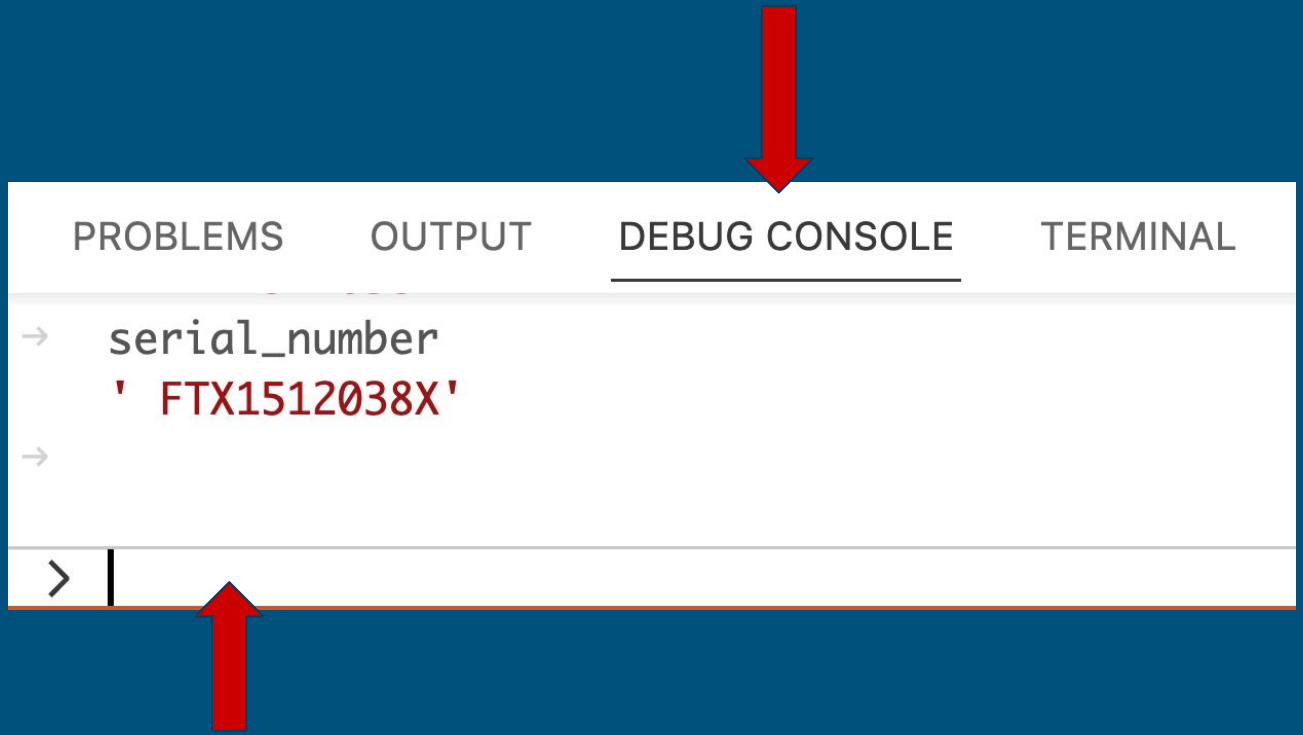
Current Location



Debugger Toolbar



Debug Console



Inspect variables

sys.path and $PYTHONPATH

```python
import sys
from rich import print

print(sys.path)
```

How does Python locate other Python Libraries?
Where does Python even look?

```
# Modify PYTHONPATH to get extra libraries
export PYTHONPATH=~/python-libs
export PYTHONPATH=$PYTHONPATH:~/DJANGOX/djproject/
```

```
>>> print(sys.path)
[
    '',
    '/Library/Frameworks/Python.framework/Versions/3.10/lib/python310.zip',
    '/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10',
    '/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/lib-dynload',
    '/Users/ktbyers/GIT/pynet-ons-oct22/.venv/lib/python3.10/site-packages'
]
>>>
```

$PYTHONPATH and VS Code

https://code.visualstudio.com/docs/python/environments#_use-of-the-pythonpath-variable

Use of the PYTHONPATH variable

The PYTHONPATH environment variable specifies additional locations where the Python interpreter should look for modules. In VS Code, PYTHONPATH can be set through the terminal settings (terminal.integrated.env.*) and/or within an `.env` file.

$PYTHONPATH and VS Code

```
    "python.terminal.executeInFileDir": true,
    "terminal.integrated.env.osx": {
        "PYTHONPATH": "${workspaceFolder}/src"
    },
    "terminal.integrated.env.windows": {
        "PYTHONPATH": "${workspaceFolder}/src"
    }
}
```

```
(.venv) $ env | grep PYT
PYTHONPATH=/Users/ktbyers/GIT/pynet-ons-oct22/src
(.venv) $
```

Libraries

```
import sys
from rich import print
from netmiko import ConnectHandler
```

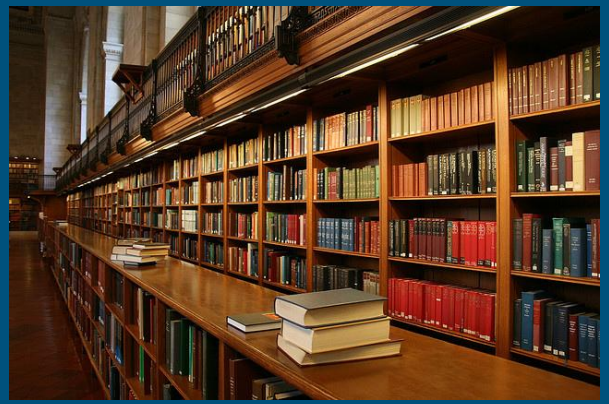


Photo: Viva Vivanista (Flickr)

PIP = Package Installer for Python


```
(.venv) $ python -m pip list
Package            Version
------------------ -------
appnope            0.1.3
astroid            2.12.11
```

pypi = Python Package Index

```
(.venv) $ python -m pip show rich
Name: rich
Version: 12.6.0
Summary: Render rich text, tables, progress bars, syntax highlighting, markdown and more to the t
nal
Home-page: https://github.com/willmcgugan/rich
Author: Will McGugan
Author-email: willmcgugan@gmail.com
License: MIT
Location: /Users/ktbyers/GIT/pynet-ons-oct22/.venv/lib/python3.10/site-packages
Requires: commonmark, pygments
Required-by: pdbr
```

PIP = Package Installer for Python


```
(.venv) $ python -m pip uninstall rich
Found existing installation: rich 12.6.0
Uninstalling rich-12.6.0:
  Would remove:
    /Users/ktbyers/GIT/pynet-ons-oct22/.venv/lib/python3.10/site-packages/rich-12.6.0.dist-info/*
    /Users/ktbyers/GIT/pynet-ons-oct22/.venv/lib/python3.10/site-packages/rich/*
Proceed (Y/n)? y
  Successfully uninstalled rich-12.6.0
```

```
(.venv) $ python -m pip install rich==12.6.0
Collecting rich==12.6.0
  Using cached rich-12.6.0-py3-none-any.whl (237 kB)
Requirement already satisfied: pygments<3.0.0,>=2.6.0 in ./.venv/lib/python3.10/site-packages (fr
ich==12.6.0) (2.13.0)
Requirement already satisfied: commonmark<0.10.0,>=0.9.0 in ./.venv/lib/python3.10/site-packages
m rich==12.6.0) (0.9.1)
Installing collected packages: rich
Successfully installed rich-12.6.0
```

PIP = Package Installer for Python

```
(.venv) $ python -m pip freeze
appnope==0.1.3
astroid==2.12.11
asttokens==2.0.8
backcall==0.2.0
black==22.10.0
```

PIP = Package Installer for Python


```
(.venv) $ python -m pip install -r ./requirements-dev.txt
Requirement already satisfied: ipython in ./.venv/lib/python3.10/site-packages
s-dev.txt (line 1)) (8.5.0)
Requirement already satisfied: pdbr in ./.venv/lib/python3.10/site-packages (fr
ev.txt (line 2)) (0.7.3)
```

```
(.venv) [ktbyers@pydev2 netmiko]$ python -m pip install -e .
Obtaining file:///home/ktbyers/netmiko
  Preparing metadata (setup.py) ... done
Requirement already satisfied: setuptools>=38.4.0 in ./.venv/lib/pyt
```

```
(.venv) [ktbyers@pydev2 netmiko]$ pip list | grep netmiko
netmiko                         4.1.2     /home/ktbyers/netmiko
```

Python Linters

Pylint or pycodestyle

Consistency and conventions make your life easier.

Finds obvious errors. Finds problems you might not be aware of.

pylint my_file.py
pycodestyle my_file.py
pylama my_file.py

Auto formatting with Python Black

Python Linters and VS Code

```
{
    "python.linting.enabled": true,
    "python.formatting.provider": "black",
    "python.formatting.blackPath": "black",
    "python.linting.pycodestyleEnabled": true,
    "python.linting.pycodestylePath": "pycodestyle",
    "python.linting.pycodestyleArgs": [
        "--max-line-length=100"
    ],
    "editor.formatOnSave": true,
}
```

Classes and Objects - Why?

```
class MyClass:

    def __init__(self, arg1, arg2, arg3):
        self.arg1 = arg1
        self.arg2 = arg2
        self.arg3 = arg3
```

python™

Classes and Objects - Why? Why? Why?

Code Reuse - But Why Aren't Functions Enough (or are they)?

```
def send_command(remote_conn, cmd):
    """Send a command down the telnet channel."""
    cmd = cmd.rstrip()
    remote_conn.write(cmd + "\n")
    time.sleep(1)
    return remote_conn.read_very_eager()

def login(remote_conn, username, password):
    """Login to network device."""
    output = remote_conn.read_until("sername:", TELNET_TIMEOUT)
    remote_conn.write(username + "\n")
    output += remote_conn.read_until("ssword:", TELNET_TIMEOUT)
    remote_conn.write(pas
    return output

def disable_paging(remote_conn, paging_cmd="terminal length 0"):
    """Disable the paging of output (i.e. --More--)"""
    return send_command(remote_conn, paging_cmd)
```

remote_conn

remote_conn

remote_conn

Code Reuse - But Why Aren't Functions Enough (or are they)?

Classes and Objects - Why? Why? Why?

Situations where you have a lot of common code, but you want to overwrite/replace some subset of behaviors.

```
class ArubaSSH(CiscoSSHConnection):
    """Aruba OS support"""

    def __init__(self, **kwargs: Any) -> None:
        if kwargs.get("default_enter") is None:
            kwargs["default_enter"] = "\r"
        # Aruba has an auto-complete on space behavior that is problematic
        if kwargs.get("global_cmd_verify") is None:
            kwargs["global_cmd_verify"] = False
        return super().__init__(**kwargs)
```

python™

Classes and Objects - Don't Miss the Forest for all the Trees!

Code Reuse - But Why Aren't Functions Enough (or are they)?

Classes and Objects - Syntax

class keyword

ClassName (PascalCase)

The "init" method

```
class MyClass:

    def __init__(self, arg1, arg2, arg3):
        self.arg1 = arg1
        self.arg2 = arg2
        self.arg3 = arg3
```

python™

Classed and Objects - What the heck is going on with "init"

class keyword

The "init" method

```
class MyClass:

    def __init__(self, arg1, arg2, arg3):
        self.arg1 = arg1
        self.arg2 = arg2
        self.arg3 = arg3
```

Think of it as a blueprint for what happens when you create instances of these things.

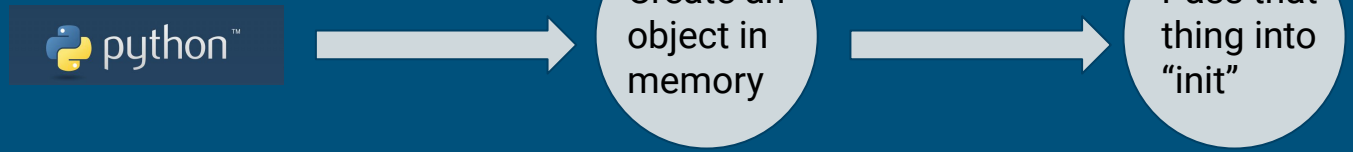
python™

Our Blueprint

```python
class NetworkDevice:

    def __init__(self, host, username, password):
        self.host = host
        self.username = username
        self.password = password


rtr1 = NetworkDevice(
            host="cisco3.lasthop.io",
            username="cisco",
            password="cisco"
        )
```

Create one of these NetworkDevice things.

python → Create an object in memory → Pass that thing into “init”



But we need a way to refer to that thing in our code: Enter "self"

python™

That thing in memory

The rest of these are just normal parameters (just like functions)

```
class NetworkDevic

    def __init__(self, host, username, password):
        self.host = host
        self.username = username
        self.password = password


rtr1 = NetworkDevice(
            host="cisco3.lasthop.io",
            username="cisco",
            password="cisco"
        )
```

A very common pattern is to assign the things you pass in to the object

When we create the object, we don't have to refer to the object (Python will create it automatically).

python™

Why?

A very common pattern is to assign the things you pass in to the object.

```
 10 rtr1 = NetworkDevice(
 11             host="cisco3.lasthop.io",
 12             username="cisco",
 13             password="cisco"
 14         )
 15
 16
❱ 17 import pdbr; pdbr.set_trace()
 18
(Pdbr) rtr1
<__main__.NetworkDevice object at 0x717bccb9ca10>
(Pdbr) rtr1.host
'cisco3.lasthop.io'
(Pdbr) rtr1.username
'cisco'
(Pdbr) rtr1.password
'cisco'
```

We created an instance of this thing.

The thing now knows the attributes that we bound to it (in __init__).

Classes - Exercise1

Create a NetworkDevice class. The class should have fields for ip_addr, username, and password.

Assign the ip_addr, username, and password field to the object in the “init” method.

Create four different network device objects using this class.

For one of these objects print out the ip_addr, username, and password attributes of the object.

Exercises:
./day3/py_classes/classes_ex1.txt

Sure, okay, but what does this allow us to do?

python™

```
class TelnetConn:
    """Establish and manage telnet connection to network devices."""

    def __init__(self, ip_addr, username, password):
        self.ip_addr = ip_addr
        self.username = username
        self.password = password
        try:
            self.remote_conn = telnetlib.Telnet(
                self.ip_addr, TELNET_PORT, TELNET_TIMEOUT
            )
        except socket.timeout:
            sys.exit("Connection timed-out")

    def login(self):
        """Login to network device."""
        output = self.remote_conn.read_until("sername:", TELNET_TIMEOUT)
        self.remote_conn.write(self.username + "\n")
        output += self.remote_conn.read_until("ssword:", TELNET_TIMEOUT)
        self.remote_conn.write(self.password + "\n")
        time.sleep(1)
        return output
```

We don't need to pass the arguments any longer (they are bound to the object)

The telnet connection is bound to the object.

We can then reuse it later.

One Last Thing

```
class TelnetConn:
    """Establish and manage telnet connection to network devices."""

    def __init__(self, ip_addr, username, password):
        self.ip_addr = ip_addr
        self.username = username
        self.password = password
        try:
            self.remote_conn = telnetlib.Telnet(
                self.ip_addr, TELNET_PORT, TELNET_TIMEOUT
            )
        except socket.timeout:
            sys.exit("Connection timed-out")

    def login(self):
        """Login to network device."""
        output = self.remote_conn.read_until("sername:", TELNET_TIMEOUT)
        self.remote_conn.write(self.username + "\n")
        output += self.remote_conn.read_until("ssword:", TELNET_TIMEOUT)
        self.remote_conn.write(self.password + "\n")
        time.sleep(1)
        return output
```

This is a method. It is just a fancy name for a function inside of an object.

python™

One Other Last Thing

Functions work pretty well.

We are all trying to learn and apply new things.

Try to get a toehold.

Classes - Exercise2

Expand on exercise 1 and create two new methods for your NetworkDevice class.

Method1 should be named “print_ip” and should print the IP address of the device.

Method2 should be named “print_creds” and should print the username and password of the device.

Both methods should have only a single parameter “self”. No other parameters should be used.

Create an instance of this class and call your two methods.

Exercises:
./day3/py_classes/classes_ex2.txt

More Data Structures

Nesting Lists:

```
my_list = [
    ["rtr1", "rtr2", "sw1", "sw2"],
    ["22.17.1.1", "22.17.1.2", "22.17.1.20", "22.17.1.21"],
    ["sj1", "sj1", "sj1", "sj1"],
]
```

```
In [4]: print(my_list)
[
    ['rtr1', 'rtr2', 'sw1', 'sw2'],
    ['22.17.1.1', '22.17.1.2', '22.17.1.20', '22.17.1.21'],
    ['sj1', 'sj1', 'sj1', 'sj1']
]
```

More Data Structures

Nesting Lists:

```
In [4]: print(my_list)
[
    ['rtr1', 'rtr2', 'sw1', 'sw2'],
    ['22.17.1.1', '22.17.1.2', '22.17.1.20', '22.17.1.21'],
    ['sj1', 'sj1', 'sj1', 'sj1']
]
```

```
print("First List:")
print(my_list[0])
print()
```

```
First List:
['rtr1', 'rtr2', 'sw1', 'sw2']
```

```
print("Second List, Element 0")
print(my_list[1][0])
print()
```

```
Second List, Element 0
22.17.1.1
```

Nested Dictionaries

```
nested_dict = {
    "rtr1": {"ip_addr": "22.17.1.1", "vendor": "cisco", "platform": "ios-xe"},
    "rtr2": {"ip_addr": "22.17.1.2", "vendor": "cisco", "platform": "ios-xe"},
    "sw1": {"ip_addr": "22.17.1.20", "vendor": "aruba", "platform": "aruba-cx"},
    "sw2": {"ip_addr": "22.17.1.21", "vendor": "aruba", "platform": "aruba-cx"},
}
```

Nested Dictionaries

```
Nested Dictionary:
{
    'rtr1': {'ip_addr': '22.17.1.1', 'vendor': 'cisco', 'platform': 'ios-xe'},
    'rtr2': {'ip_addr': '22.17.1.2', 'vendor': 'cisco', 'platform': 'ios-xe'},
    'sw1': {'ip_addr': '22.17.1.20', 'vendor': 'aruba', 'platform': 'aruba-cx'},
    'sw2': {'ip_addr': '22.17.1.21', 'vendor': 'aruba', 'platform': 'aruba-cx'}
}
```

```
print()
print("RTR2 Entry")
print(nested_dict["rtr2"])
print()
```

```
RTR2 Entry
{'ip_addr': '22.17.1.2', 'vendor': 'cisco', 'platform': 'ios-xe'}
```

Drilling in More

```
Nested Dictionary:
{
    'rtr1': {'ip_addr': '22.17.1.1', 'vendor': 'cisco', 'platform': 'ios-xe'},
    'rtr2': {'ip_addr': '22.17.1.2', 'vendor': 'cisco', 'platform': 'ios-xe'},
    'sw1': {'ip_addr': '22.17.1.20', 'vendor': 'aruba', 'platform': 'aruba-cx'},
    'sw2': {'ip_addr': '22.17.1.21', 'vendor': 'aruba', 'platform': 'aruba-cx'}
}
```

```
print("SW1 Entry, Platform Field:")
print(nested_dict["sw1"]["platform"])
print()
```

```
SW1 Entry, Platform Field:
aruba-cx
```

Complex Data Structures - Exercise1

Create a data structure that is a list of dictionaries.

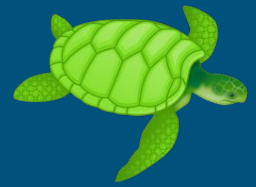
The list should have four elements. Each of the elements should be a dictionary with four keys: device_name, ip_addr, vendor, and platform.

Print out the last dictionary using rich.print.

Print out the device_name and ip_addr fields from the first dictionary.

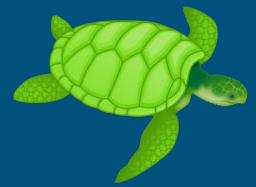
Exercises:
./day3/complex_data_struct/struct_ex1.txt

Data structures all the way down



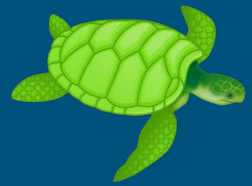
```
In [5]: print(data)
{
    '_meta': {
        'int_vlan': {
            '_mappings': {
                'root': 'int_vlan',
                'key_list': {
                    'id': 'id',
                    'int_vlan_shut': 'int_vlan_shut',
                    'int_vlan_ip.ipparams': 'int_vlan_ip.ipparams',
                    'int_vlan_ip.ipaddr': 'int_vlan_ip.ipaddr',
                    'int_vlan_ip.ipmask': 'int_vlan_ip.ipmask',
                    'int_vlan_ip.dhcp-client': 'int_vlan_ip.dhcp-client',
                    'int_vlan_ip.client-id': 'int_vlan_ip.client-id',
                    'int_vlan_ip.cid': 'int_vlan_ip.cid',
```

Step down into the data structure: Layer-by-layer



```
In [16]: data.keys()
Out[16]: dict_keys(['_meta', '_data'])

In [17]: type(data["_data"])
Out[17]: dict

In [18]: data["_data"].keys()
Out[18]: dict_keys(['int_vlan'])
```

Step down into the data structure: Layer-by-layer



Dictionaries: Look at the Keys

Lists: Look at the List length and potentially a single element.

```
In [16]: data.keys()
Out[16]: dict_keys(['_meta', '_data'])

In [17]: type(data["_data"])
Out[17]: dict

In [18]: data["_data"].keys()
Out[18]: dict_keys(['int_vlan'])
```

```
In [15]: data = data["_data"]

In [16]: type(data)
Out[16]: dict

In [17]: data.keys()
Out[17]: dict_keys(['int_vlan'])
```

Step down into the data structure: Layer-by-layer

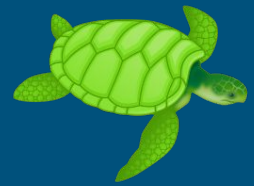
Dictionaries: Look at the Keys

Lists: Look at the List length and potentially a single element.

```
In [18]: data = data["int_vlan"]

In [19]: type(data)
Out[19]: list

In [20]: len(data)
Out[20]: 5
```

```
In [21]: data[0]
Out[21]:
{'id': 95,
 'int_vlan_ip': {'ipaddr': '95.95.1.1',
  'ipparams': 'ipaddrmask',
  'ipmask': '255.255.255.0'},
```

Step down into the data structure: Layer-by-layer

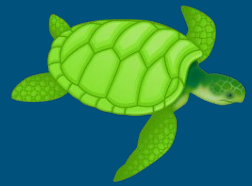


Dictionaries: Look at the Keys

Lists: Look at the List length and potentially a single element.

```
In [30]: data[0].keys()
Out[30]: dict_keys(['id', 'int_vlan_ip', 'int_vlan_routing', 'int_vlan_ndra_hl
imit', 'int_vlan_ndra_interval', 'int_vlan_ndra_ltime', 'int_vlan_ndra_mtu', '
int_vlan_nd_reachtime', 'int_vlan_nd_rtrans_time', 'int_vlan_mtu', 'int_vlan_s
uppress_arp'])
```

Step down into the data structure: Layer-by-layer



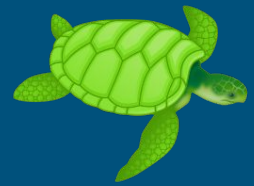
Dictionaries: Look at the Keys

Lists: Look at the List length and potentially a single element.

```
In [32]: data[0]["int_vlan_ip"]
Out[32]: {'ipaddr': '95.95.1.1', 'ipparams': 'ipaddrmask', 'ipmask': '255.255.
255.0'}
```

```
In [33]: data[0]["int_vlan_ip"]["ipaddr"]
Out[33]: '95.95.1.1'
```

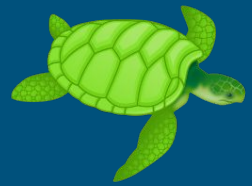
Step down into the data structure: Layer-by-layer



But we are still only looking at one element of this list. How to handle ALL of the elements?

```
for element in data:
    print(element["int_vlan_ip"]["ipaddr"])
```

Step down into the data structure: Layer-by-layer



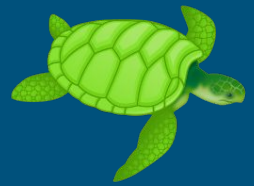
This didn't work–why not?

```
for element in data:
    print(element["int_vlan_ip"]["ipaddr"])
```

```
      1 for element in data:
----> 2     print(element["int_vlan_ip"]["ipaddr"])

KeyError: 'ipaddr'
```

```
In [40]: element["int_vlan_ip"]
Out[40]: {'dhcp-client': True, 'ipparams': 'dhcp_opt'}
```

Step down into the data structure: Layer-by-layer



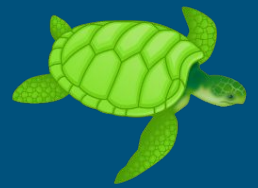
What do we do about it?

```
In [44]: for element in data:
    ...:     int_vlan_ip = element.get("int_vlan_ip", {})
    ...:     ip_addr = int_vlan_ip.get("ipaddr")
    ...:     print(ip_addr)
    ...:
95.95.1.1
None
None
None
None
```

Remember Your Process

Peel the data structure back layer-by-layer

1. Determine the type of the data structure (probably either a list or a dictionary).
2. If dictionary, look at the keys (hopefully, you can pick one that you want).
3. If a list, look at the length of the list. If length is one, just unwrap the list and repeat this process.
4. If the list is longer than one, look at one single element and see if you can determine what to do from there.
5. With lists, it is possible you need to use a loop to handle all of the elements uniformly.



Complex Data Structures - Exercise2

Load the JSON file "struct_data1.json" in a Python script as a Python data structure. This text represents the routing table on a Cisco switch.

Use rich.print to print this object to stdout so you can get a good idea what you're dealing with.

Print the type and length of the object. In this scenario, we know that all of the elements of this object are of the same type and length, but this is not always the case.

Print the type and length of the zeroith element of the object to be sure what data types you are working with.

Exercises:
./day3/complex_data_struct/struct_ex2.txt

Complex Data Structures - Exercise2

Create a new dictionary variable called "parsed_data".

Iterate through the structured data, creating a key in the "parsed_data" dictionary for every network that is NOT of "protocol" "L" (local routes).

Add the "nexthop_if" and "nexthop_ip" values to this dictionary. Print your output using “rich” when complete, it should look similar to this:

```
{
    '0.0.0.0': {
        'nexthop_interface': 'Vlan3967',
        'nexthop_ip': '172.31.255.254'
    },
    '172.31.254.0': {
        'nexthop_interface': 'Vlan254',
        'nexthop_ip': ''
    },
    '172.31.255.5': {
        'nexthop_interface': 'Loopback0',
        'nexthop_ip': ''
    },
    '172.31.255.254': {
        'nexthop_interface': 'Vlan3967',
        'nexthop_ip': ''
    }
}
```

Exercises:
./day3/complex_data_struct/struct_ex2.txt

See you in Chicago.