Getting Started: Cisco UCS Python SDK for UCS Manager Lab

This guide is intended to provide an introduction to the Cisco UCS Python SDK, which is the toolkit developed to manage and automate UCS operations using the Python scripting language.

Introduction:
Cisco Unified Computing System (UCS) is a converged computing platform designed for running the leading enterprise applications, virtualization platforms, and operating systems. Cisco UCS Manager is embedded management solution within the UCS that provides a single point of management of the UCS infrastructure. Through a 100% open and documented XML API, Cisco UCS has been integrated with many major systems management tools commonly found in heterogeneous data centers.

This class is intended to provide an introduction to the Cisco UCS Python SDK, which is a toolkit developed to manage and automate UCS operations using the Python scripting language.

Tools used in this lab:

- **Cisco UCS Python SDK** (**Python modules for managing Cisco UCS)–** UCS Python SDK is a set of Python modules, each containing one or more classes, developed specifically to automate UCS Manager. UCS Python SDK is developed with PEP8 compliance and supports every Managed Object exposed by the UCS XML Model UCS. A special feature (convert_to_ucs_python), enables automatic Python script generation from operations performed in the UCS Manager GUI.
- **UCS Platform Emulator (UCSPE)** – UCSPE is a virtual machine that was developed to enable the use of Cisco UCS Manager and the UCS XML API without requiring physical hardware. UCSPE significantly shortens the development cycle for applications that are based on the UCS XML API. You can create and test programs using only UCSPE installed on a laptop. UCSPE presents a controlled environment for emulating large-scale UCS environments, changes in the hardware inventory, firmware upgrade testing, troubleshooting real UCS problems, testing UCS XML API requests in lieu of a real server, invoking the UCS GUI to become familiar with its usage and features, invoking the model object browser and internal documentation, and providing a convenient way to demonstrate UCS operations for training and other activities.

UCS Python SDK for UCS Manager Lab

Table of Contents:

Introduction to UCS Python SDK Part I

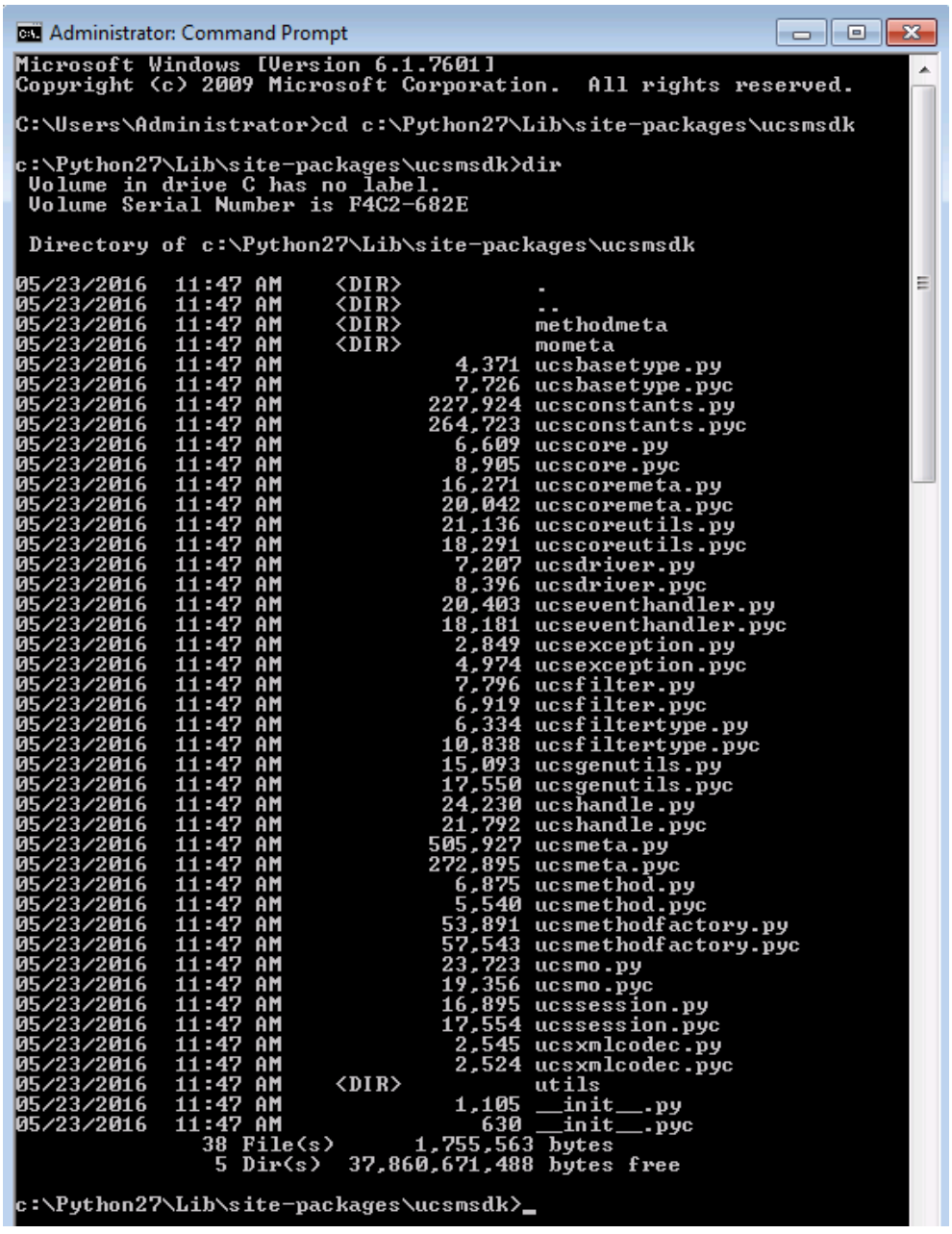
Example #1a: Explore – list modules / packages in Python SDK

On the Windows Desktop, Python 2.7 has been pre-installed, along with the Cisco UCS Python SDK.

- Official Documentation for the UCS Python SDK : https://ciscoucs.github.io/ucsmsdk_docs/

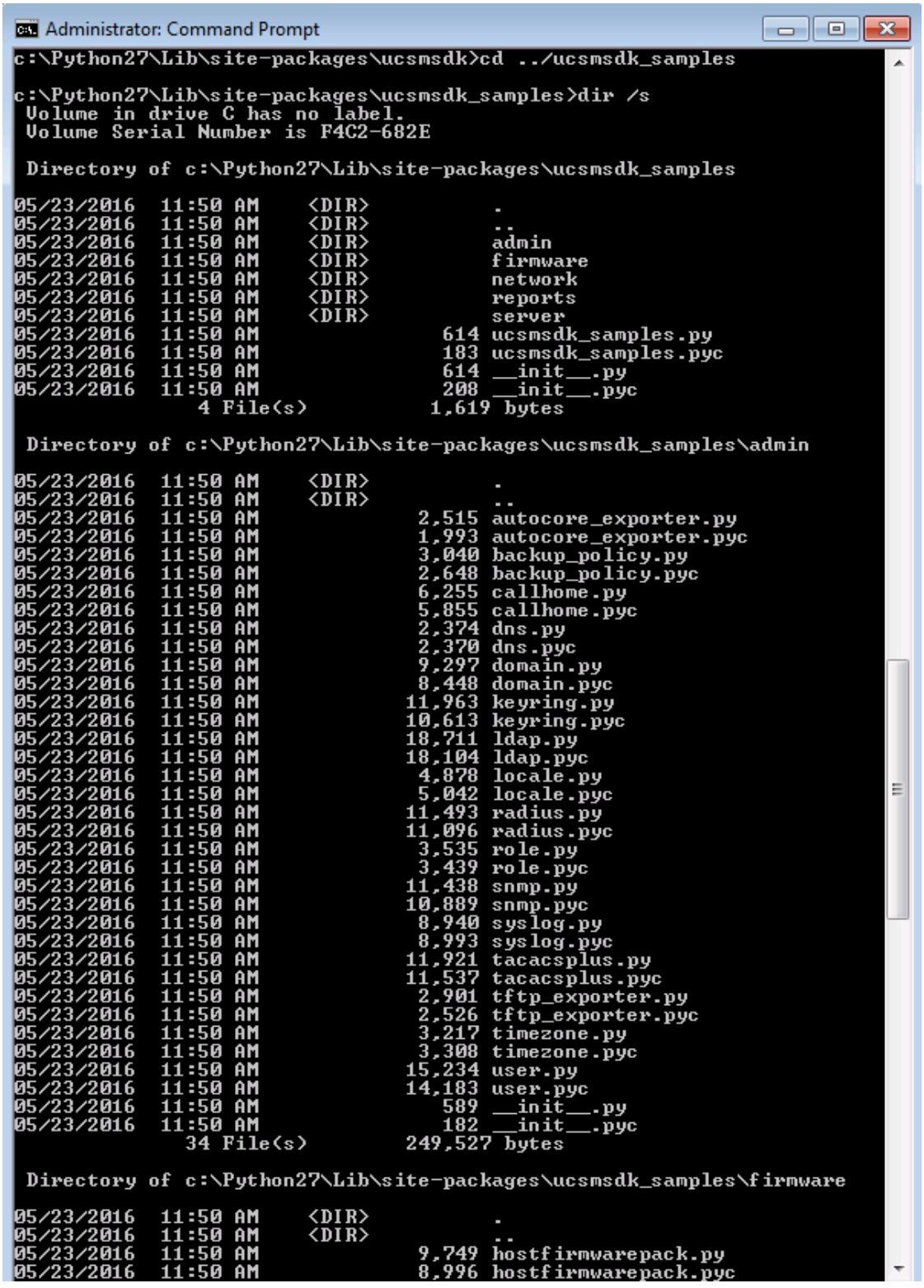
Open Windows File Manager or a Command prompt session to locate and view the Python modules and associated python scripts available with the Cisco UCS Python SDK
(note: all third party Python modules are typically located in the subdirectory: \Lib\site-packages\).

```
cd c:\Python27\Lib\site-packages\ucsmsdk
dir
```



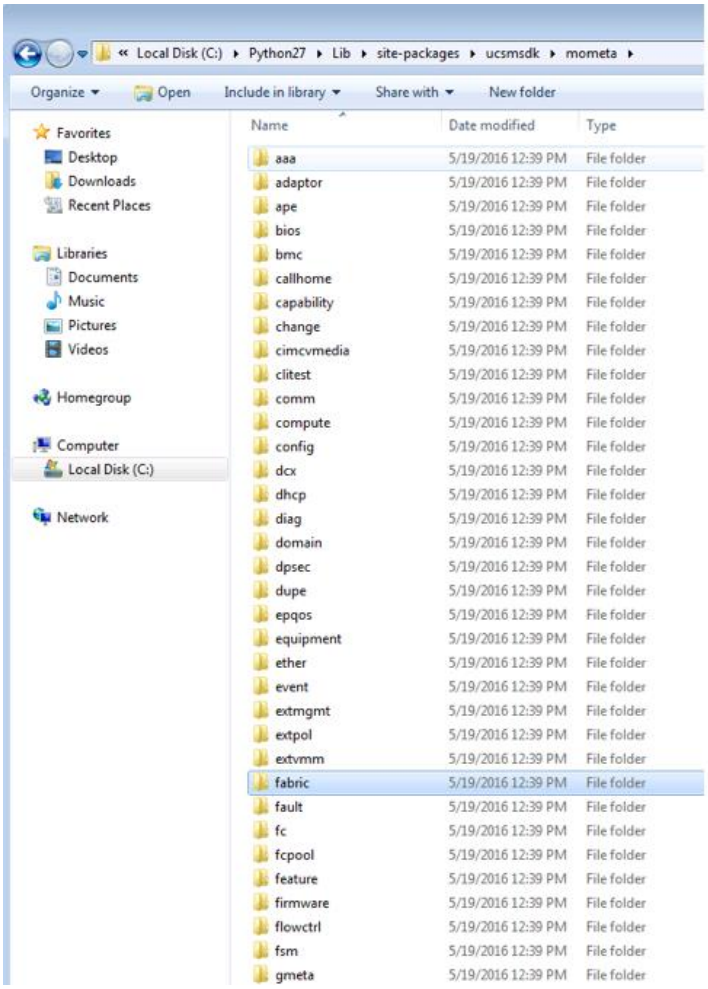
In addition to the core Python Modules, the SDK includes a comprehensive set of samples, grouped in subfolders by their use/function (ex: admin, firmware, network, reports, server)

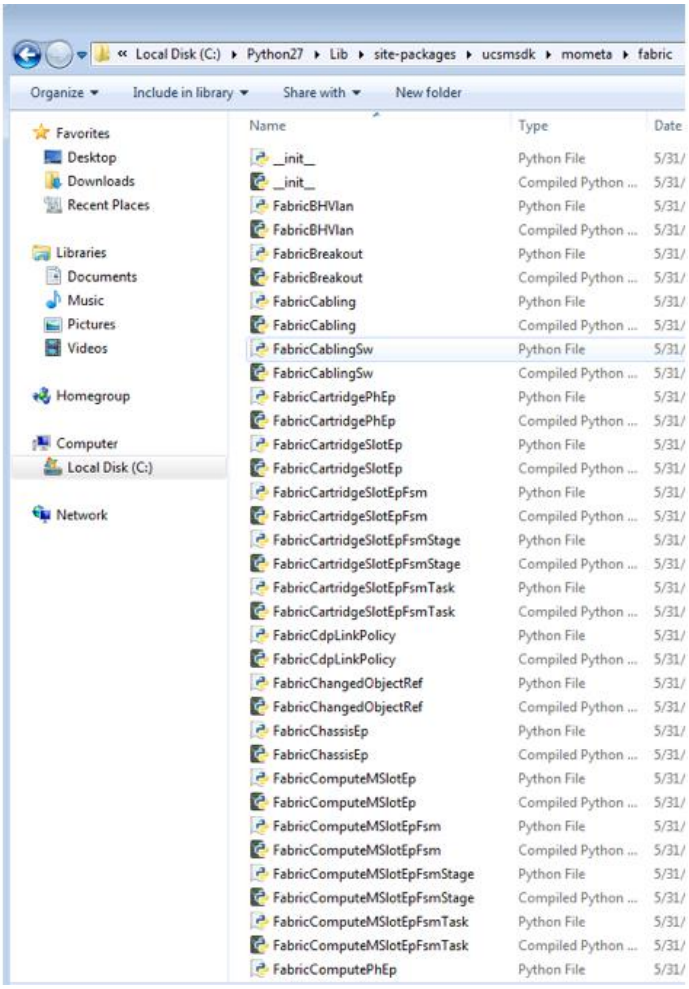
```
cd ../ucsmsdk_samples
dir /s
```



Example #1b: Explore – list classes / managed objects in the UCS Python SDK

Under the ucsmsdk directory, you will find the "mometa" directory (Managed Object metadata). If you look inside any of the subdirectories (ex: fabric), you will see modules/scripts that map to all the classes available in the UCS Manager XML Model:



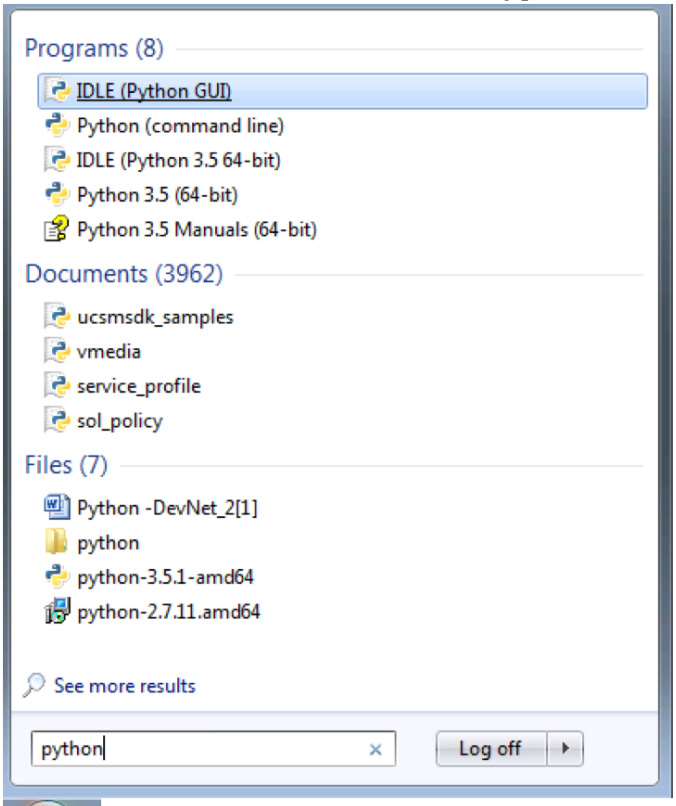


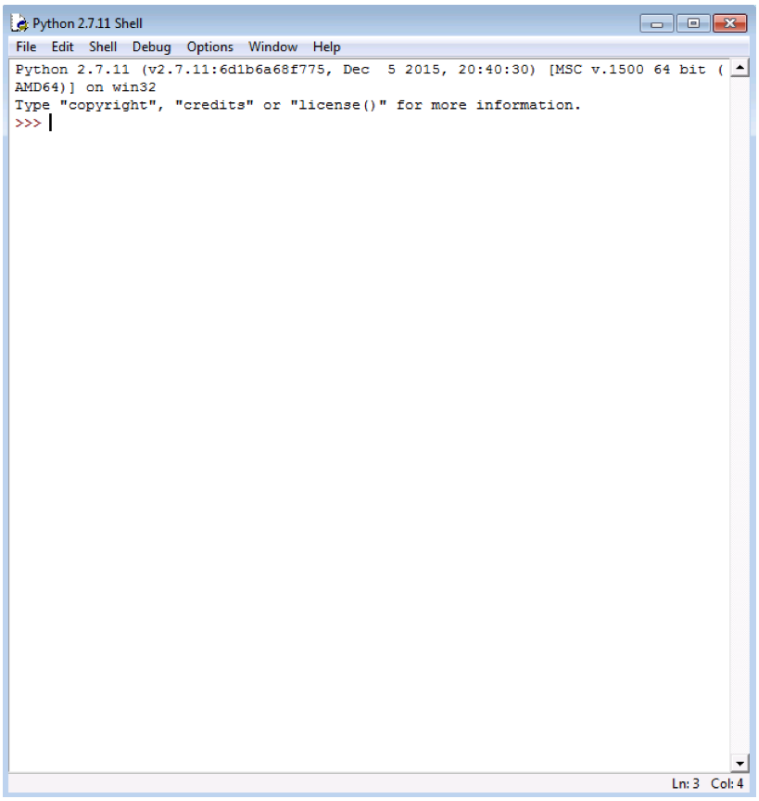
The UCS Manager Python SDK abstracts these low level modules/scripts.

- Ultimately, as administrators use the UCSM Python SDK, these low level modules/scripts are being leveraged to create/modify/delete objects in the UCS Manager XML Model.

Example #1c: Launch Python Interactive Shell (IDLE)

Click the Windows Start button and type "**`python`**" to search for the Python IDLE editor:





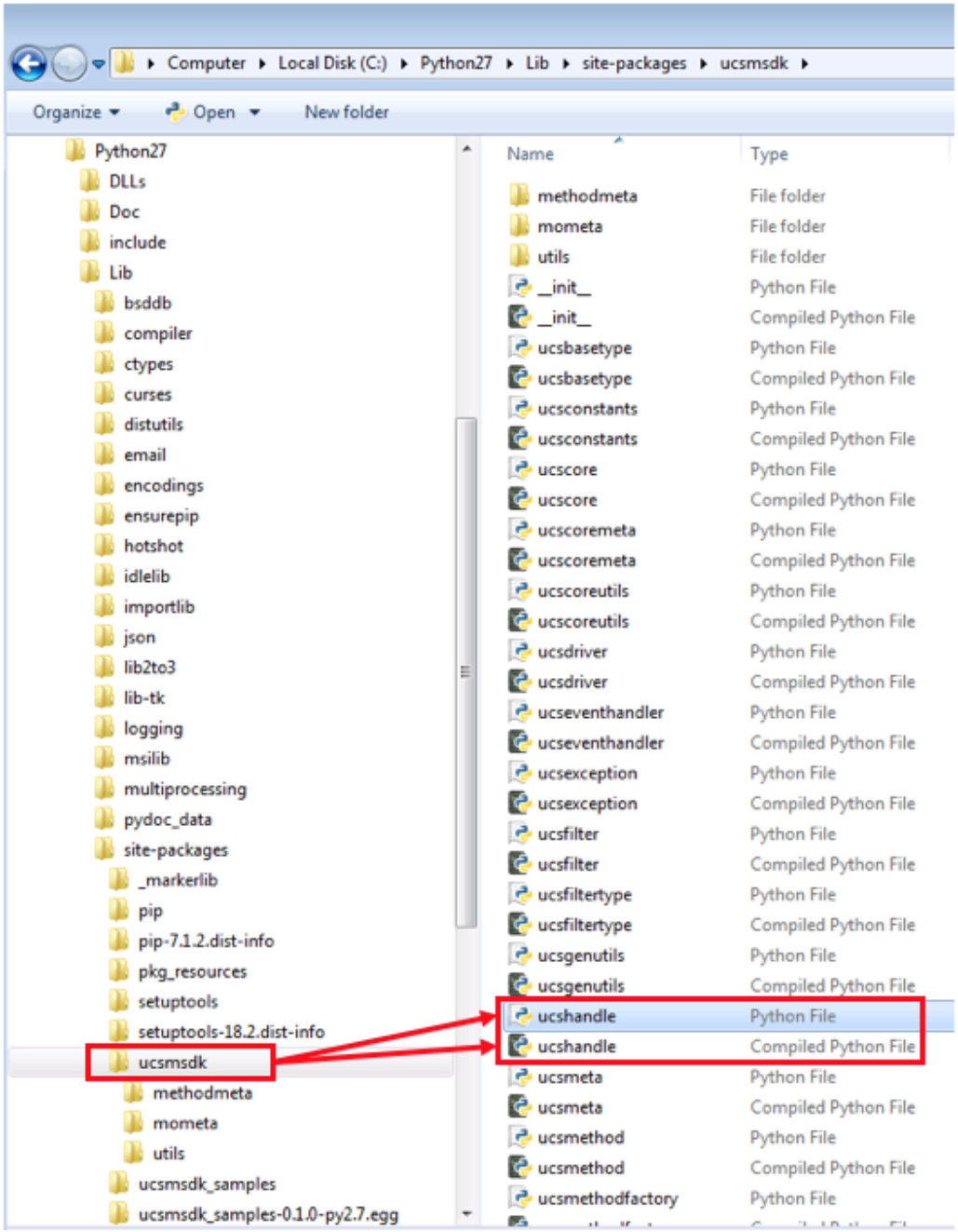
Example #1d: Getting help

To begin using the SDK, users will need to import classes into a running Python shell environment.

- As mentioned earlier, all third party-modules are located under Lib\site-packages, and so you will see the ucsmsdk and ucsmsdk_samples directories below in C:\Python27\Lib\site-packages\

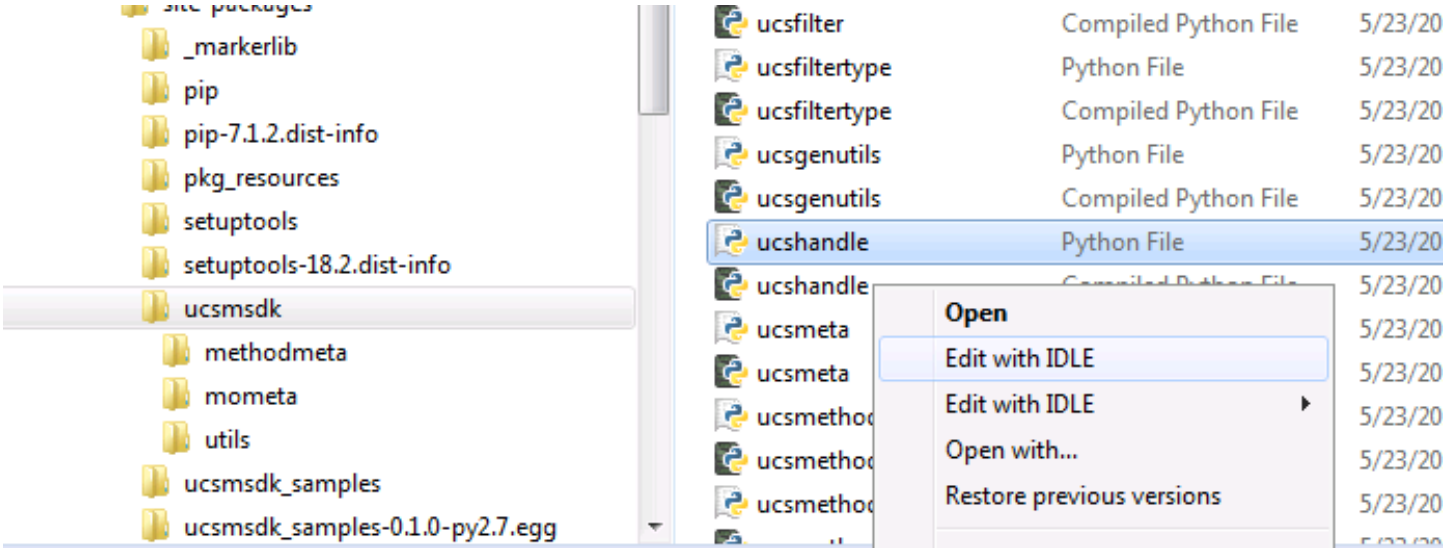
In this example, we will explore the ucshandle module, view associated classes/functions and then view the available help data.

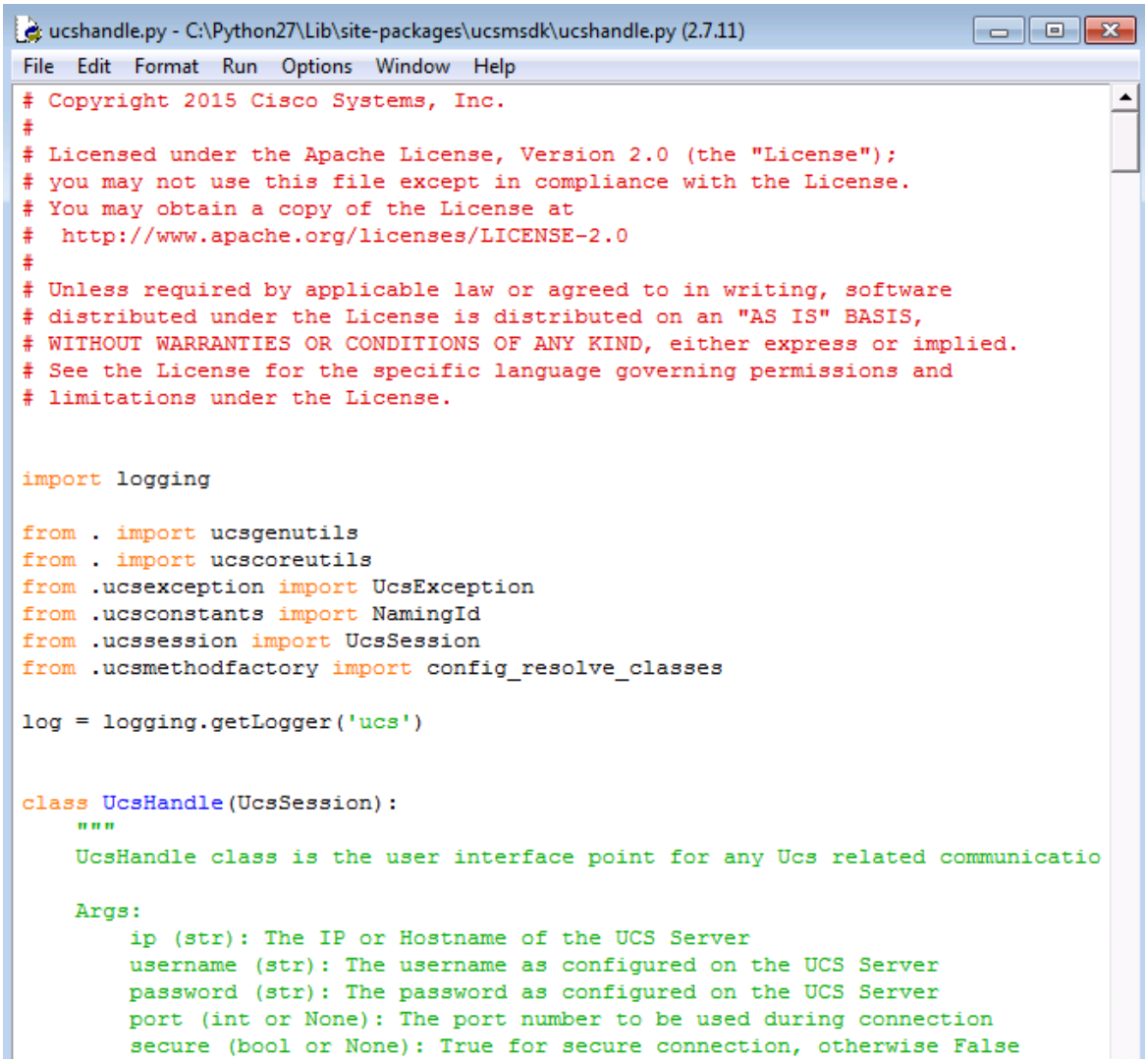
Notice that there is a Compiled Python file and Python file (script) associated with each module:



To use the "ucshandle" module, we will need to import the module, but let's first open the ucshandle Python File (script).

We can do this by right clicking on the ucshandle Python file, and selecting to open/edit this file in the IDLE Editor:





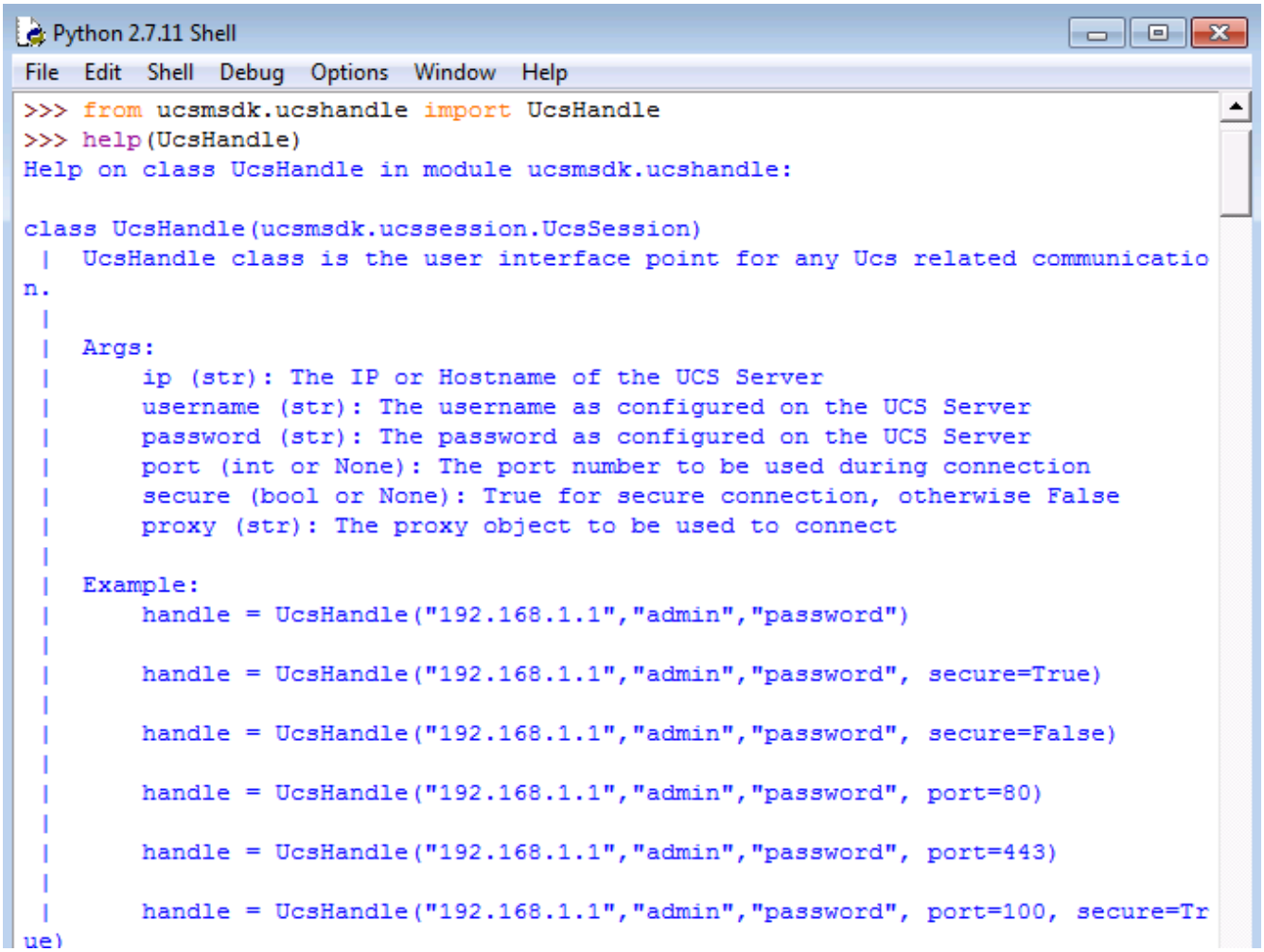
You will find that in this modules defines the “UcsHandle” Class.
To import this module, and then view the available help, use the following usage:

```
from <directory>.<module_name> import <Class or Function>
```

In this example, we are going to import the “UcsHandle” function from the “ucshandle” module and view the help.

To view the help of the “ucshandle” module (used to login/logout of a UCS Domain), run the following:

```
from ucsmsdk.ucshandle import UcsHandle
help(UcsHandle)
```



The help information will provide information on all available arguments, including use examples.

Example #1e: Connecting / Disconnecting from UCS Manager:

In this next example, we will use the UcsHandle class (from the ucshandle module) to connect to a UCS Domain. The UCS Platform Emulator (UCSPE) will access ANY login credentials.

Run the following commands to authenticate against the UCS Manager XML API, and establish a session.

- Note: the **class or function name** (derived from help) needs to be specified and is **case sensitive**

handle = UcsHandle("<ip/hostname of UCSM>", "<login_name>", "<password>" ; then handle.login()

```
handle = UcsHandle("10.0.254.12", "admin", "passsword")
handle.login()
```

```
>>> handle = UcsHandle("10.0.254.12", "admin", "passsword")
>>> handle.login()
True
>>>
```

Students can use the vars() function to view the new handle object, and view the various properties and attributes of this class.

- This will display all the UCSM Session information (UCS Domain name, user privileges, session cookie, cookie refresh period):

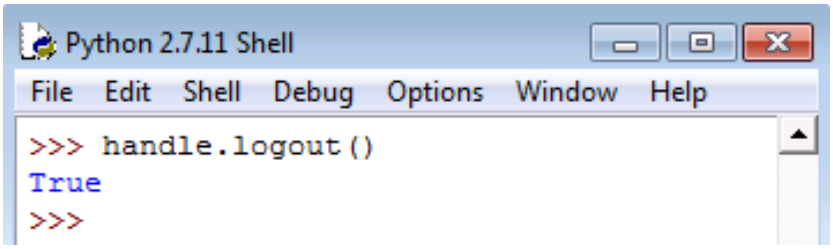
```
vars(handle)
```

```
>>> vars(handle)
{'_UcsSession__domains': 'org-root', '_UcsSession__refresh_period': '600', '_Ucs
Session__name': '', '_UcsSession__evt_channel': 'noencssl', '_UcsSession__uri':
'https://10.0.254.12:443', '_UcsSession__ip': '10.0.254.12', '_UcsSession__port'
: 443, '_UcsSession__priv': 'admin,read-only', '_UcsSession__last_update_time':
'Wed Jun 01 18:12:47 2016', '_UcsSession__secure': True, '_UcsSession__virtual_i
pv4_address': '10.0.254.12', '_UcsSession__force': False, '_UcsHandle__to_commit
': {}, '_UcsSession__ucs': '10.0.254.12', '_UcsSession__version': <ucsmsdk.ucsco
remeta.UcsVersion object at 0x03149BD0>, '_ucs': 'UCSPE-10-0-254-12', '_UcsSessi
on__dump_xml': False, '_UcsSession__channel': 'noencssl', '_UcsSession__session_
id': '', '_UcsSession__redirect': False, '_UcsSession__cookie': '1464829965/bc14
ec4c-c839-4752-8e39-a8bce7be67e4', '_UcsSession__proxy': None, '_UcsSession__ref
resh_timer': None, '_UcsSession__password': 'passsword', '_UcsSession__driver':
<ucsmsdk.ucsdriver.UcsDriver object at 0x03121C70>, '_UcsSession__username': 'ad
min'}
>>>
```

Note: The session with UCS Manager XML API will be maintained until a handle.logout is called.

To disconnect from a UCS Domain, run the following command:

```
handle.logout()
```



Example #1f: Launch the UCS Manager Java GUI:

Next we will login back into a UCS Domain, and launch the UCS Manager JAVA GUI (via http)

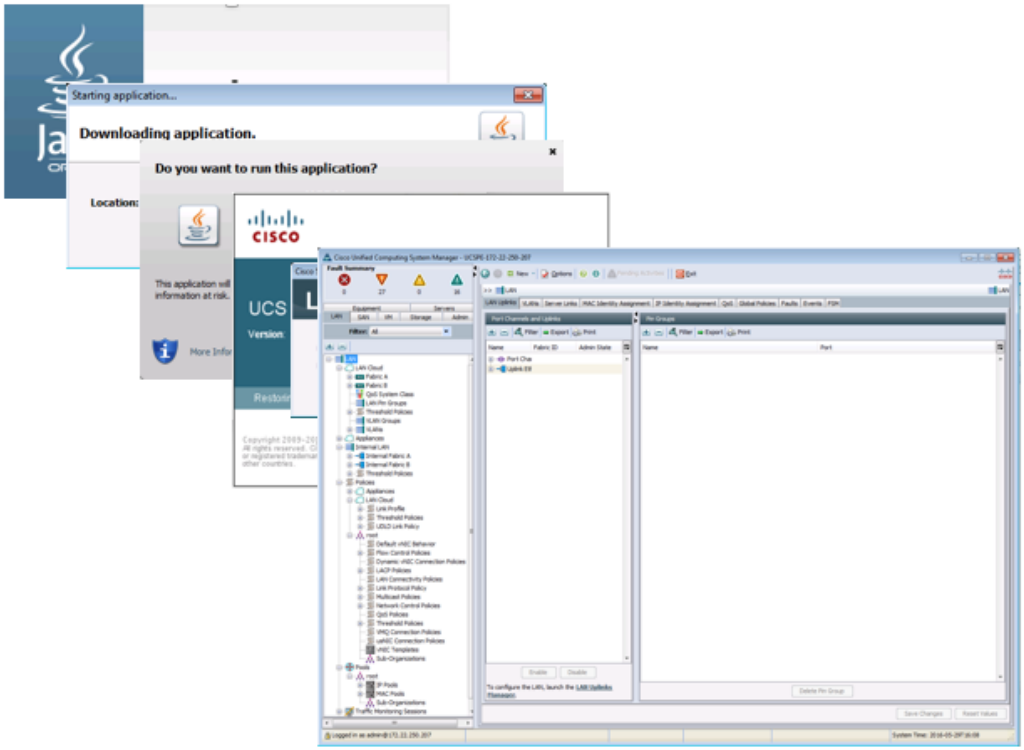
```
handle = UcsHandle("10.0.254.12","admin","passsword",secure=False)
handle.login()
```

In order to launch the UCSM Java GUI, we will be using the function "ucs_gui_launch" defined in the module "ucsguilaunch", which is located in this folder: ucsmsdk\utils directory, specified as follows:

```
from ucsmsdk.utils.ucsguilaunch import ucs_gui_launch
ucs_gui_launch(handle)
```

The UCS Manager Java file will download and launch in the local Java client:

```
>>> handle = UcsHandle("10.0.254.12","admin","passsword",secure=False)
>>> handle.login()
True
>>> from ucsmsdk.utils.ucsguilaunch import ucs_gui_launch
>>> ucs_gui_launch(handle)
2016-06-01 18:22:54,233 - ucs - DEBUG - AuthToken: <None>
2016-06-01 18:22:54,237 - ucs - DEBUG - UCSM URL: <http://10.0.254.12:80/ucsm/ucsm.jnlp>
2016-06-01 18:22:54,243 - ucs - DEBUG - javaws path: <C:\Program Files (x86)\Java\jre1.8.0_73\bin\
javaws.exe>
2016-06-01 18:22:54,319 - ucs - DEBUG - Temp Directory: <c:\users\admini~1\appdata\local\temp>
2016-06-01 18:22:54,487 - ucs - DEBUG - Java Version: <1.8.0_73>
2016-06-01 18:22:54,509 - ucs - DEBUG - Enable Log String is    <property name="jnlp.ucsm.log.show
.encrypted" value="true"/>.
>>>
```



Note: Click "Cancel" if you are presented with an "Anonymous Reporting" window.

Example #1g: Using convert_to_ucs_python() to discover Python Code:

Next we will use a special function: convert_to_ucs_python(), which has the ability to listen to the UCSM JAVA GUI logfile, identify configuration changes, and convert these to their Python Equivalent.

- A powerful, simple to use tool for discovering Python automation code.
- This function is contained in the "converttopython" module, located in the "utils" directory:

```
from ucsmsdk.utils.converttopython import convert_to_ucs_python
convert_to_ucs_python()
```

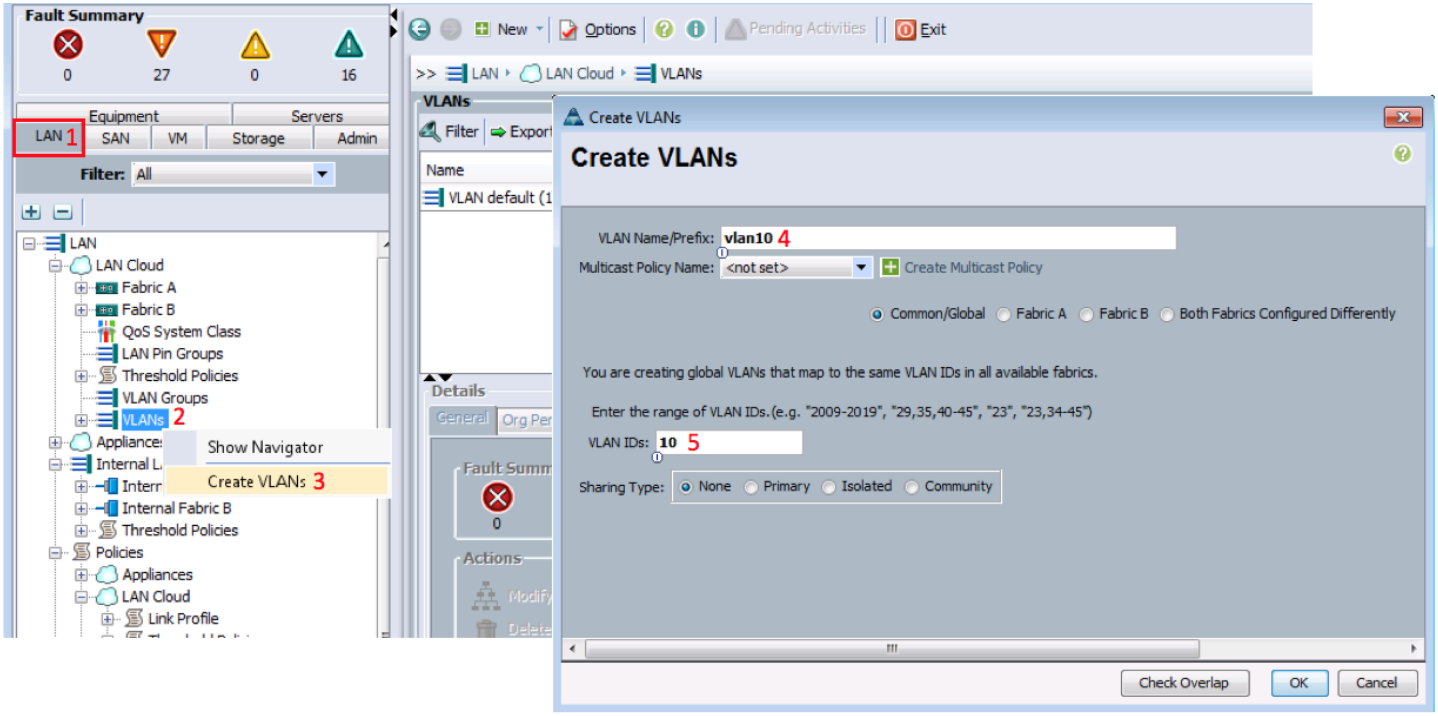
```
>>> from ucsmsdk.utils.converttopython import convert_to_ucs_python
>>> convert_to_ucs_python()
### Please review the generated cmdlets before deployment.

ucsm logfile: C:\Users\Administrator\AppData\LocalLow\Sun\Java\Deployment\log\.ucsm\centrale_3876.
log
```

Notice how the local UCSM Java GUI log file is identified above (centrale_xxxx.log)

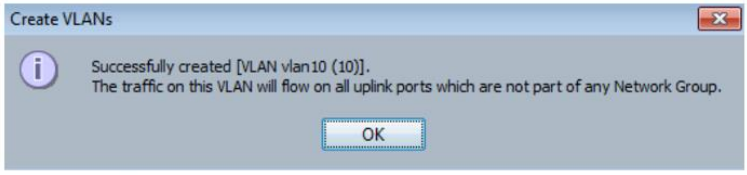
- This log file will now be monitored for any configuration created/saved via the UCSM Java GUI.
- This function will run continuously, until a "control+c" is executed to break back to the shell.

Example #1h: Create a VLAN in the GUI; discover the Python code equivalent:

In the UCSM Java GUI, navigate to the LAN Tab, right click on VLANs (under LAN Cloud) and select Create VLANs. Create "vlan10" with a VLAN ID of 10 (Steps numbered in RED):



Notice that after the VLAN is created, the Python shell displays the Python Code needed to repeat this same task. This provides a powerful method of discovering Python code to automate (anything) in UCS!




```
>>> from ucsmsdk.utils.converttopython import convert_to_ucs_python
>>> convert_to_ucs_python()
### Please review the generated cmdlets before deployment.

ucsm logfile: C:\Users\Administrator\AppData\LocalLow\Sun\Java\Deployment\log\.ucsm\centrale_3876.
log

##### Start-Of-PythonScript #####

from ucsmsdk.mometa.fabric.FabricVlan import FabricVlan

mo = FabricVlan(parent_mo_or_dn="fabric/lan", sharing="none", name="vlan10", id="10", mcast_policy
_name="", policy_owner="local", default_net="no", pub_nw_name="", compression_type="included")
handle.add_mo(mo)

handle.commit()
##### End-Of-PythonScript #####
```

Note: this output will be used in the next exercise

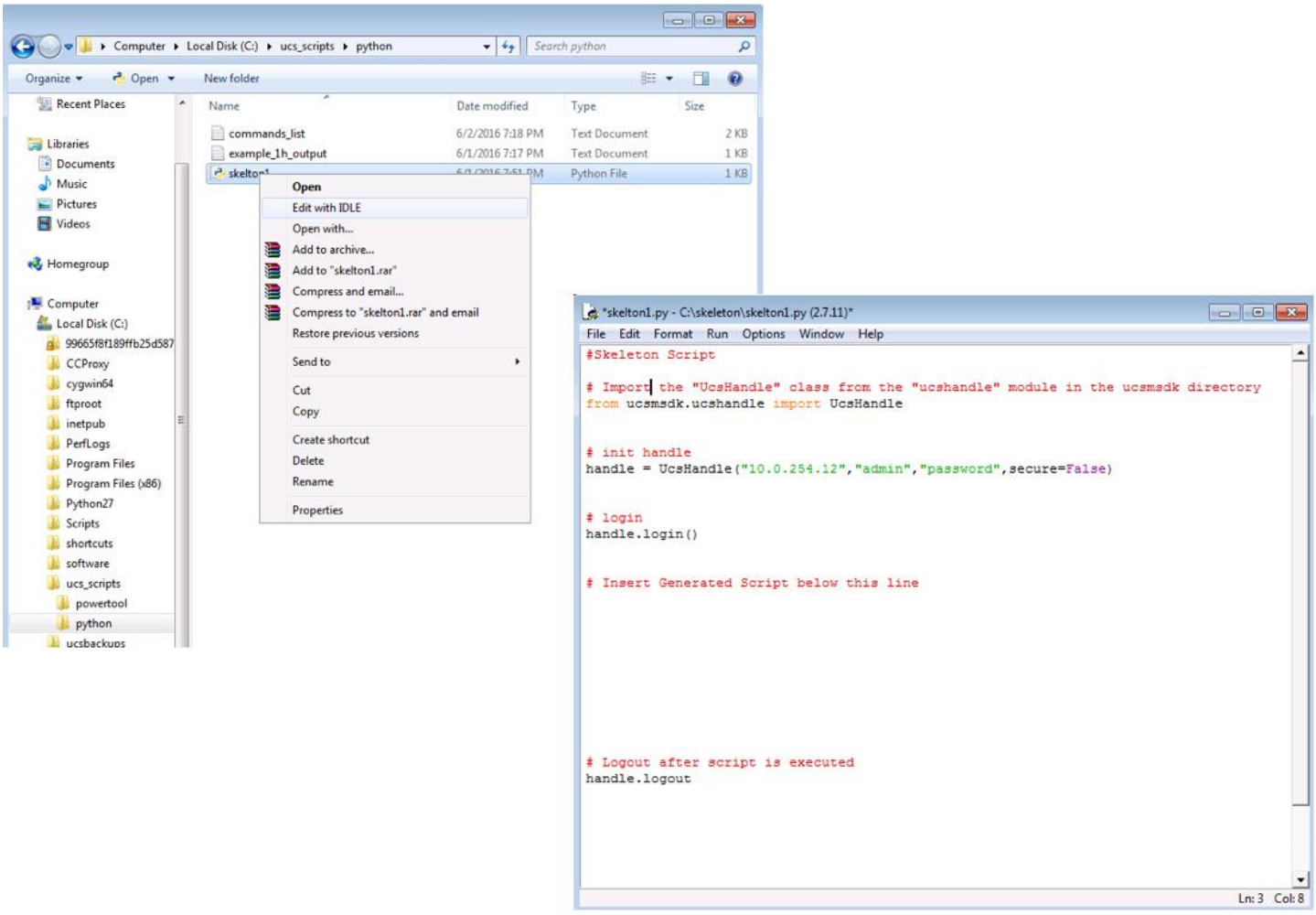
Introduction to UCS Python SDK Part II

Example #2a: Review skeleton script

For ease of executing sample code, we have provided a "skeleton" script file.

- This script will serve as an example script, and we will be pasting the code snippet discovered through the use of the convert_to_ucs_python function in example 1h.

Locate c:\ucs_scripts\python\skeleton1.py in Windows File Manager, right click and edit in the IDLE Editor:

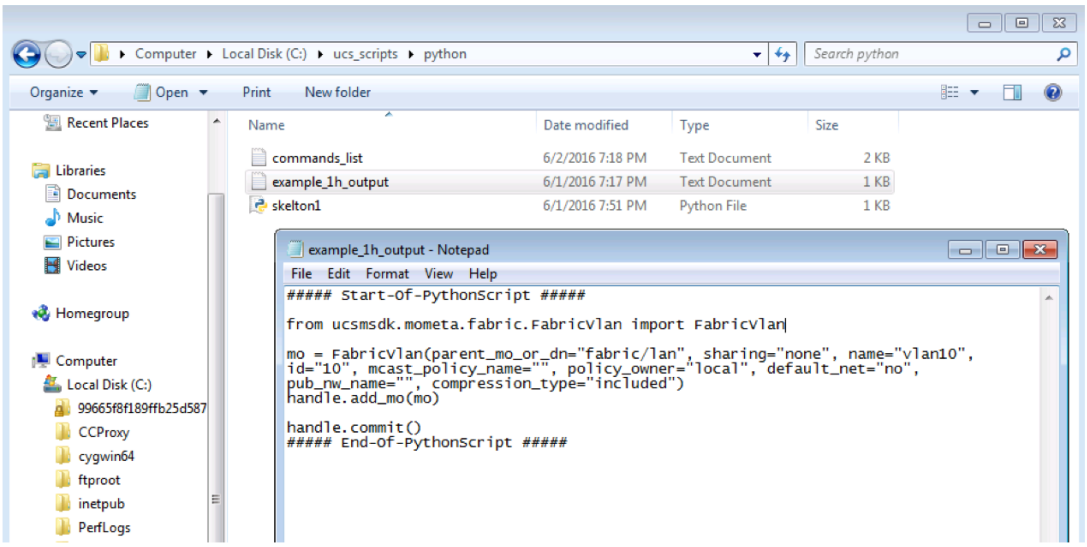


Notice that the script contains the ability to login to the UCS Domain, provides an area to paste discovered Python code, and then logs out of the UCS Domain.

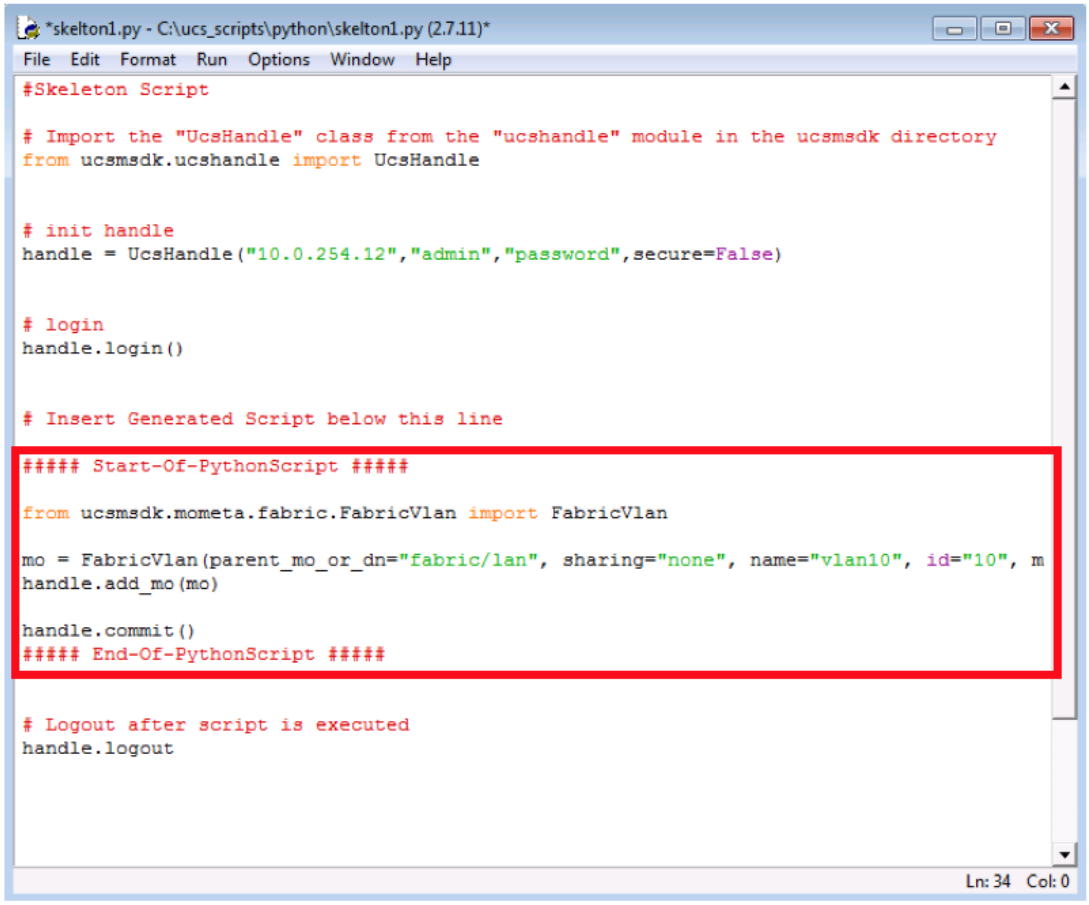
Example #2b: Copy convert_to_ucs_python() output into skeleton script

The skeleton1.py script should be open in your Python IDLE editor for this next task, and you will need the Python code discovered after exercise 1h.

- If you do not have the code output or closed the shell session, the discovered code is available in c:\ucs_scripts\python\example_1h_output



Paste the code into your skeleton1.py file (open in the IDLE editor), in the defined area:

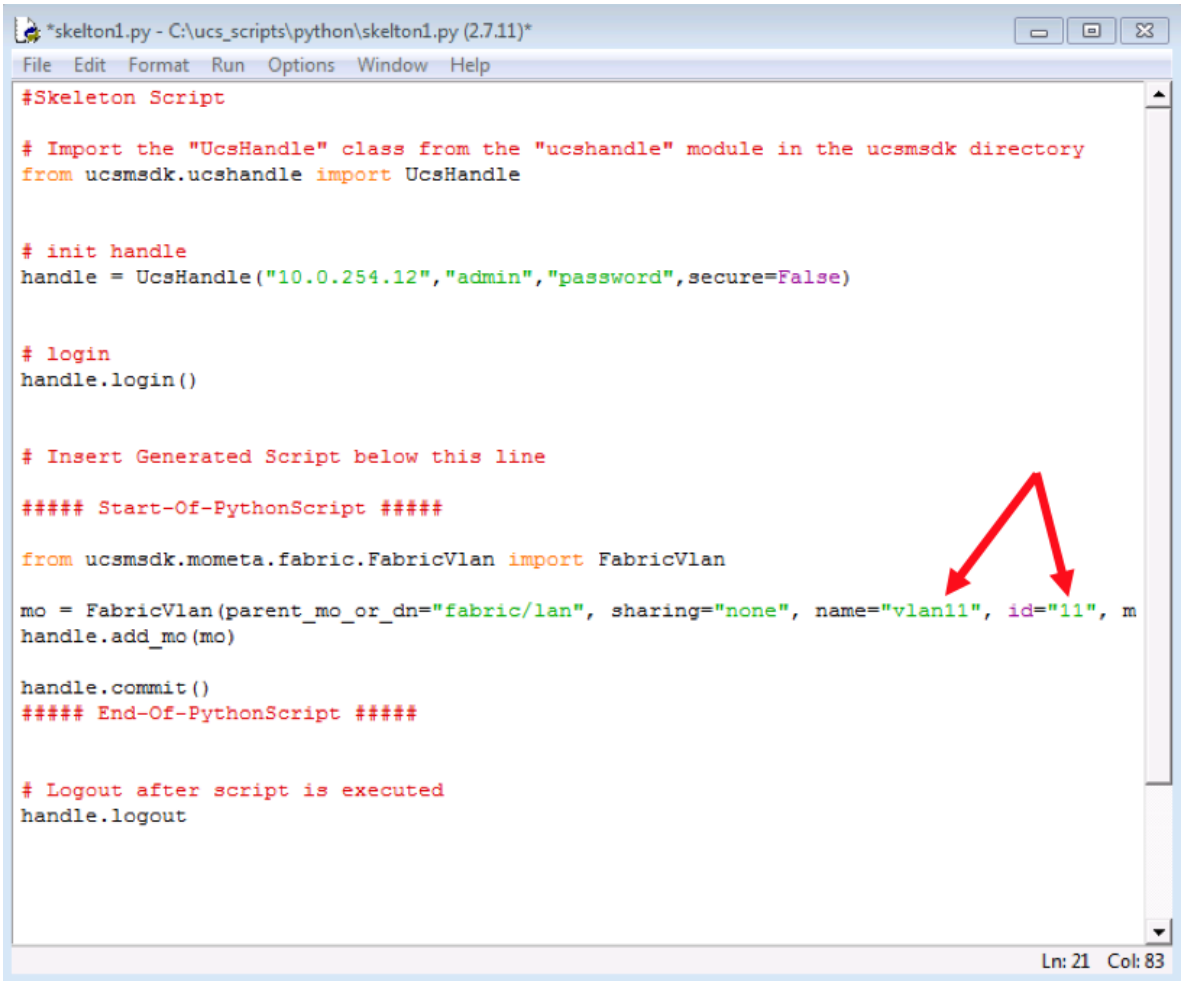


Example #2c: Edit parameters and execute script to create a new VLAN object

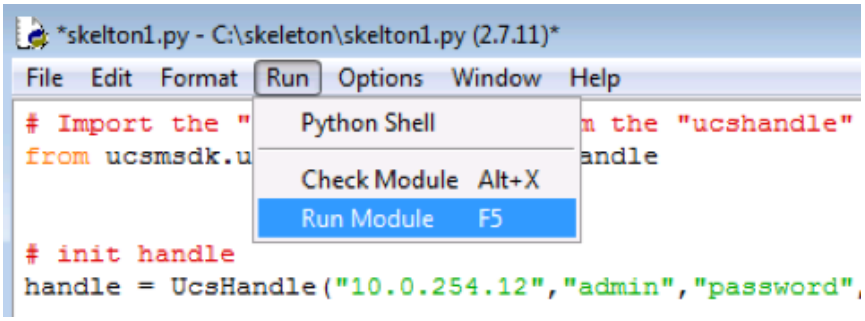
Since a VLAN named “vlan10” with ID “10” already exists in the UCS Domain (created in example 1h above), we cannot execute the script as is, or we will get an error, indicating that this object already exists.

- So we will edit some properties before executing.

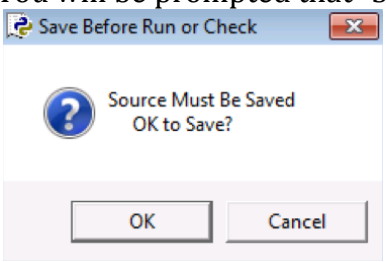
Change the VLAN Name to “vlan11” and the VLAN ID to “11”:



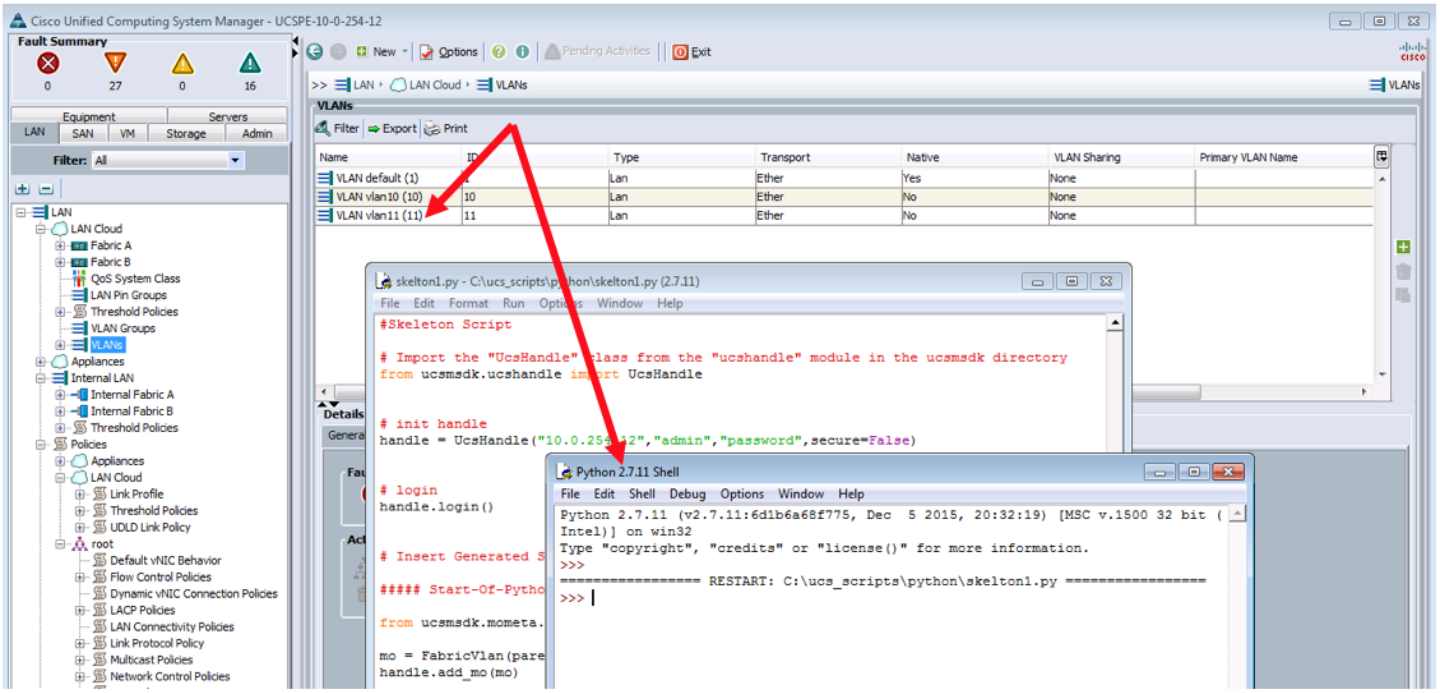
Once the edits are made, Click “Run”, then “Run Module”:



You will be prompted that “Source Must Be Saved” – Click OK:



You will then see a new Python shell spawn, containing an indication that this code was executed, and you should then see your new VLAN object appear in the UCS Manager GUI:

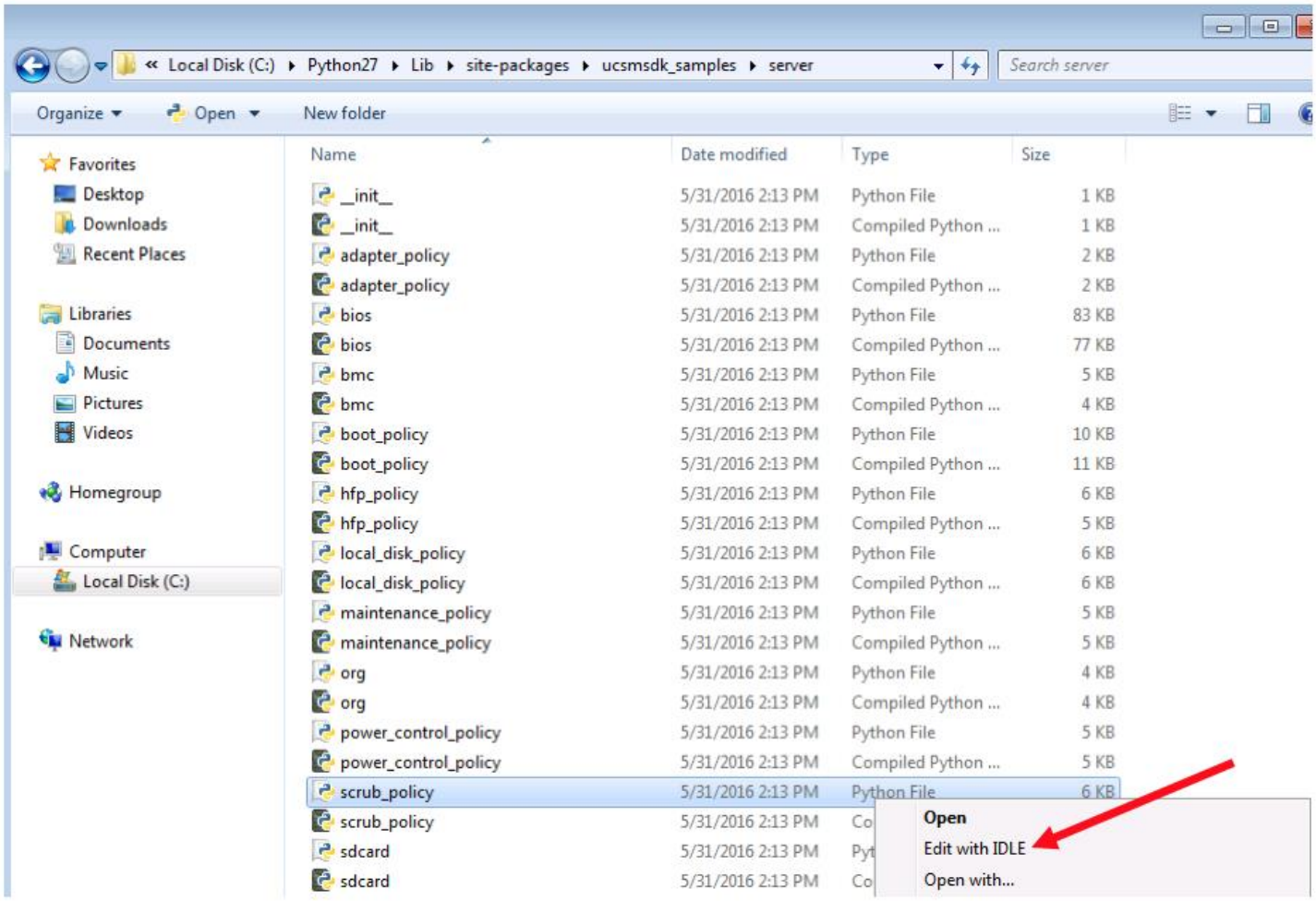


Congratulations – now you can use this methodology to learn how to automate *anything* that you create in the UCS Manager Java GUI!

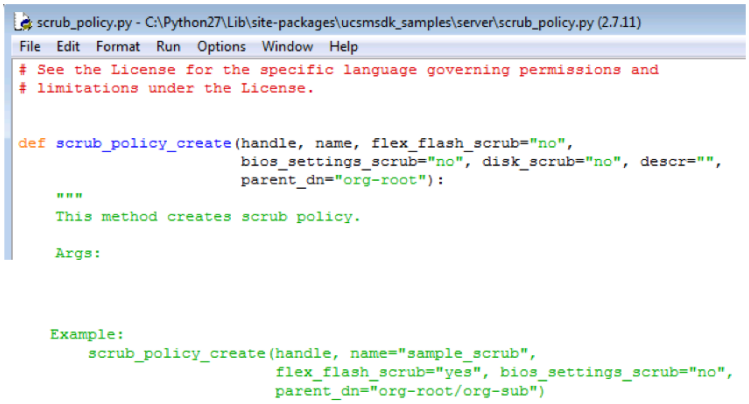
Example #2d: CREATE – create a new UCS Scrub Policy (via ucsmsdk_samples)

Next we will use the samples contained in the ucsmsdk_samples directory to create a new object, edit that object and then delete object. We will be using "scrub_policy" for these examples.

Locate the scrub_policy script, located under the ucsmsdk_samples\server directory and then right-click to "Edit in IDLE":



Notice that the function needed to create a scrub policy is "scrub_policy_create" – the script details the available arguments, and further down provides a functional example:



If your convert_to_ucs_python session is still running, issue a "control+c" to issue a KeyboardInterrupt and exit back to the Python Shell.

You can use **`vars(handle)`** to make sure you are still logged into UCS Manager – log back if necessary:

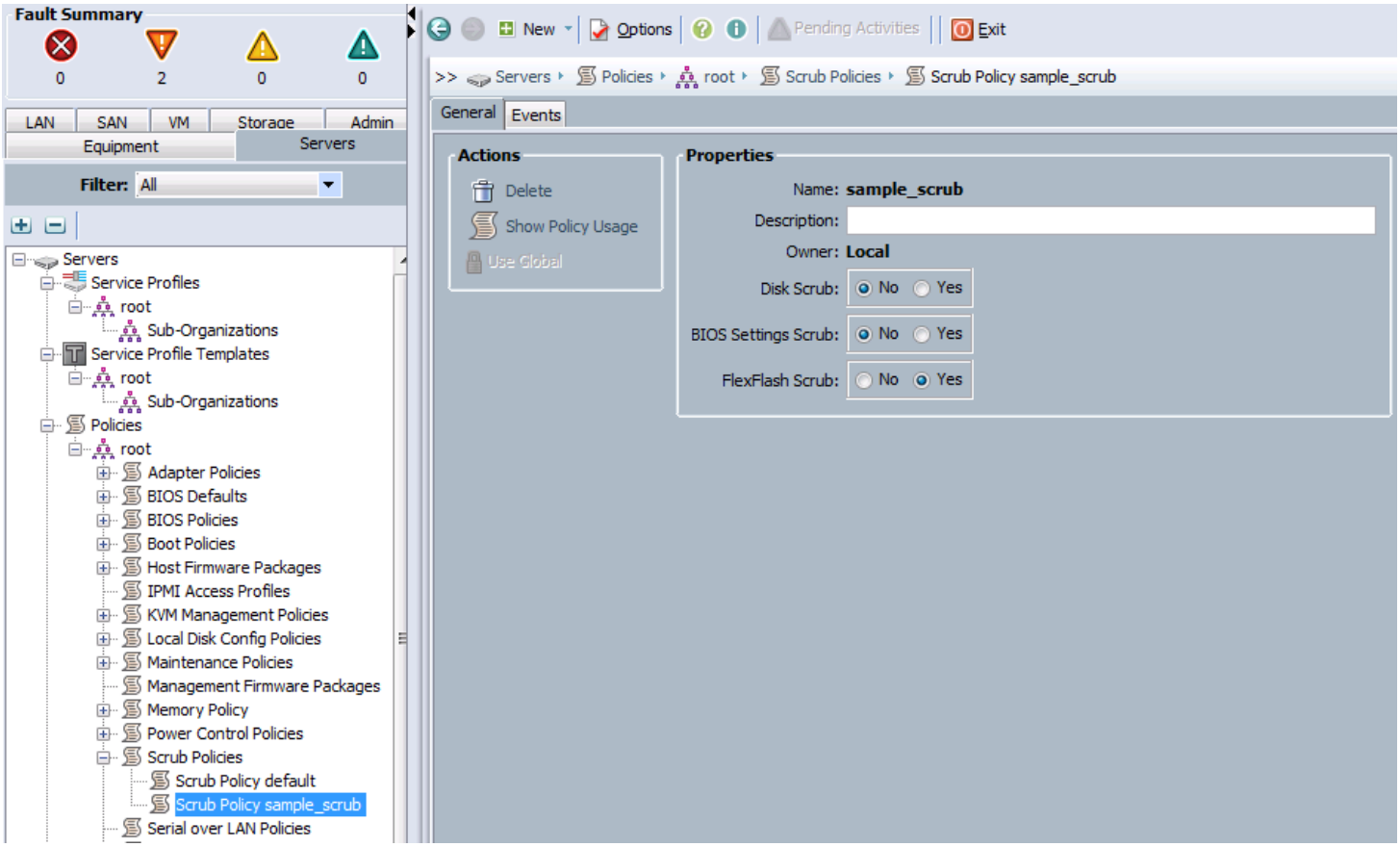
```
handle.commit()
##### End-Of-PythonScript #####

Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    convert_to_ucs_python()
  File "C:\Python27\lib\site-packages\ucsmsdk\utils\converttopython.py", line 1522, in convert_to_
ucs_python
    time.sleep(2)
KeyboardInterrupt
>>> vars(handle)
{'_UcsSession__domains': 'org-root', '_UcsSession__refresh_period': '600', '_UcsSession__name': ''
, '_UcsSession__evt_channel': 'noencssl', '_UcsSession__uri': 'http://10.0.254.12:80', '_UcsSessio
n__ip': '10.0.254.12', '_UcsSession__port': 80, '_UcsSession__priv': 'admin,read-only', '_UcsSessi
on__last_update_time': 'Wed Jun 01 18:22:38 2016', '_UcsSession__secure': False, '_UcsSession__vir
tual_ipv4_address': '10.0.254.12', '_UcsSession__force': False, '_UcsHandle__to_commit': {}, '_Ucs
Session__ucs': '10.0.254.12', '_UcsSession__version': <ucsmsdk.ucscoremeta.UcsVersion object at 0x
03168D90>, '_ucs': 'UCSPE-10-0-254-12', '_UcsSession__dump_xml': False, '_UcsSession__channel': 'n
oencssl', '_UcsSession__session_id': '', '_UcsSession__redirect': False, '_UcsSession__cookie': '1
464830555/7a42e98f-0318-4a06-b667-eef0d5b1e2f7', '_UcsSession__proxy': None, '_UcsSession__refresh
_timer': None, '_UcsSession__password': 'passsword', '_UcsSession__driver': <ucsmsdk.ucsdriver.Ucs
Driver object at 0x03168B50>, '_UcsSession__username': 'admin'}
>>>
```

In the Python shell window, execute the following commands to create a Scrub Policy named "sample-scrub" in UCSM:

```
from ucsmsdk_samples.server.scrub_policy import scrub_policy_create

scrub_policy_create(handle, name="sample_scrub",flex_flash_scrub="yes",
bios_settings_scrub="no",parent_dn="org-root")
```

```
>>> from ucsmsdk_samples.server.scrub_policy import scrub_policy_create

>>> scrub_policy_create(handle, name="sample_scrub",flex_flash_scrub="yes", bios_settings_scrub="n
o",parent_dn="org-root")

<ucsmsdk.mometa.compute.ComputeScrubPolicy.ComputeScrubPolicy object at 0x03476E50>
>>>
```

You should see the new object in the UCS Manager GUI:



Example #2e: MODIFY – make a configuration change to the scrub policy

Review the scrub_policy.py script and locate the function to modify:

scrub_policy.py - C:\Python27\Lib\site-packages\ucsmsdk_samples\server\scrub_policy.py (2.7.11)

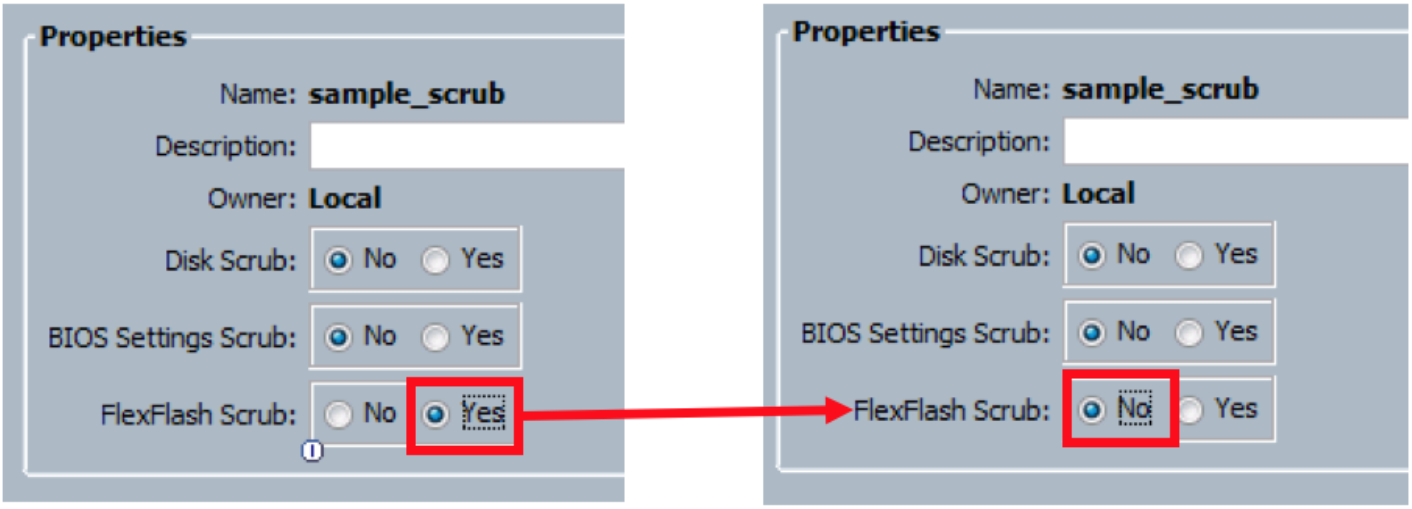
File Edit Format Run Options Window Help

```
    handle.commit()
    return mo


def scrub_policy_modify(handle, name, flex_flash_scrub=None,
                        bios_settings_scrub=None, disk_scrub=None, descr=None,
                        parent_dn="org-root"):
    """
    This method modify scrub policy.

    Args:
        handle (UcsHandle)
        name (string): Name of the scrub policy.
        flex_flash_scrub (string): "yes" or "no"
        bios_settings_scrub (string): "yes" or "no"
        disk_scrub (string): "yes" or "no"
        descr (string): Basic description.
        parent_dn (string): Parent of Org.

    Returns:
        ComputeScrubPolicy: Managed Object

    Raises:
        ValueError: If ComputeScrubPolicy is not present

    Example:
        scrub_policy_modify(handle, name="sample_scrub",
                            flex_flash_scrub="yes", bios_settings_scrub="no",
                            parent_dn="org-root/org-sub")
```

Run the following commands to modify the new scrub policy and change the "flex_flash_scrub" property from "yes" to "no" – you should see the change in the UCS Manager GUI as well:

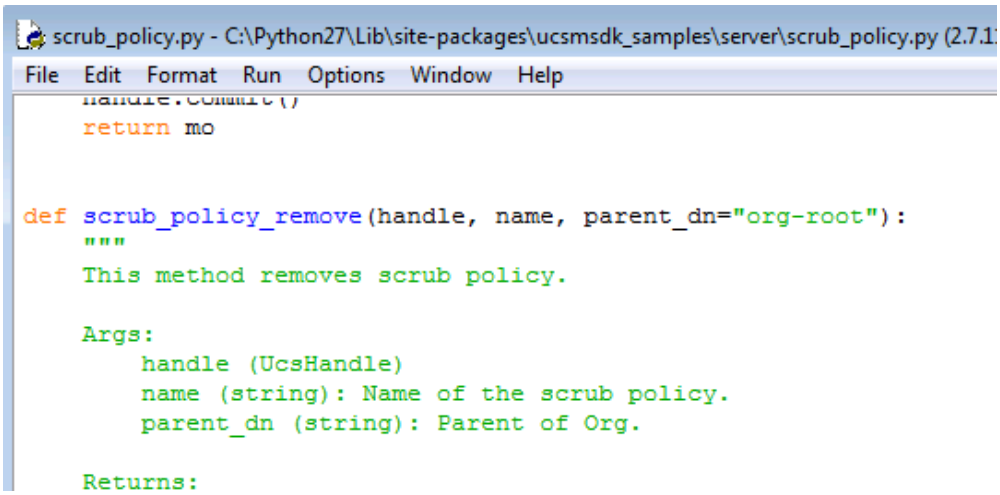
```
from ucsmsdk_samples.server.scrub_policy import scrub_policy_modify

scrub_policy_modify(handle,name="sample_scrub",flex_flash_scrub="no",parent_dn="org-root")
```

```
>>> from ucsmsdk_samples.server.scrub_policy import scrub_policy_modify
>>> scrub_policy_modify(handle,name="sample_scrub",flex_flash_scrub="no",parent_dn="org-root")
<ucsmsdk.mometa.compute.ComputeScrubPolicy.ComputeScrubPolicy object at 0x03485E50>
>>>
```



Example #2f: DELETE – delete the scrub policy

Review the scrub_policy.py script and locate the function to delete/remove:

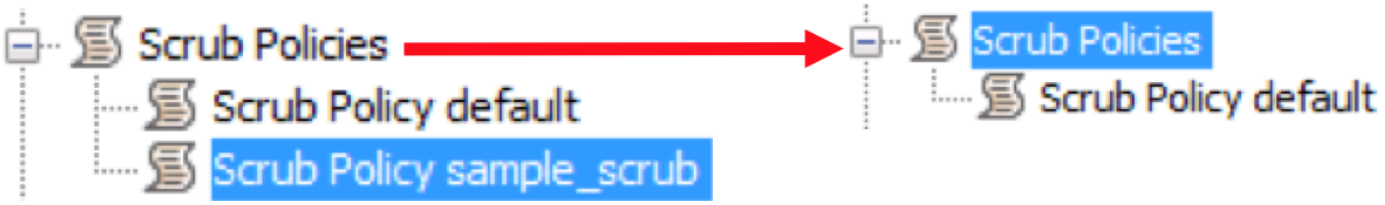


Execute the following commands to remove/delete the scrub policy:

```
from ucsmsdk_samples.server.scrub_policy import scrub_policy_remove

scrub_policy_remove(handle,name="sample_scrub",parent_dn="org-root")
```

```
>>> from ucsmsdk_samples.server.scrub_policy import scrub_policy_remove

>>> scrub_policy_remove(handle,name="sample_scrub",parent_dn="org-root")
>>>
```

The object will be removed from the UCS Manager model (and UCS Manager GUI):



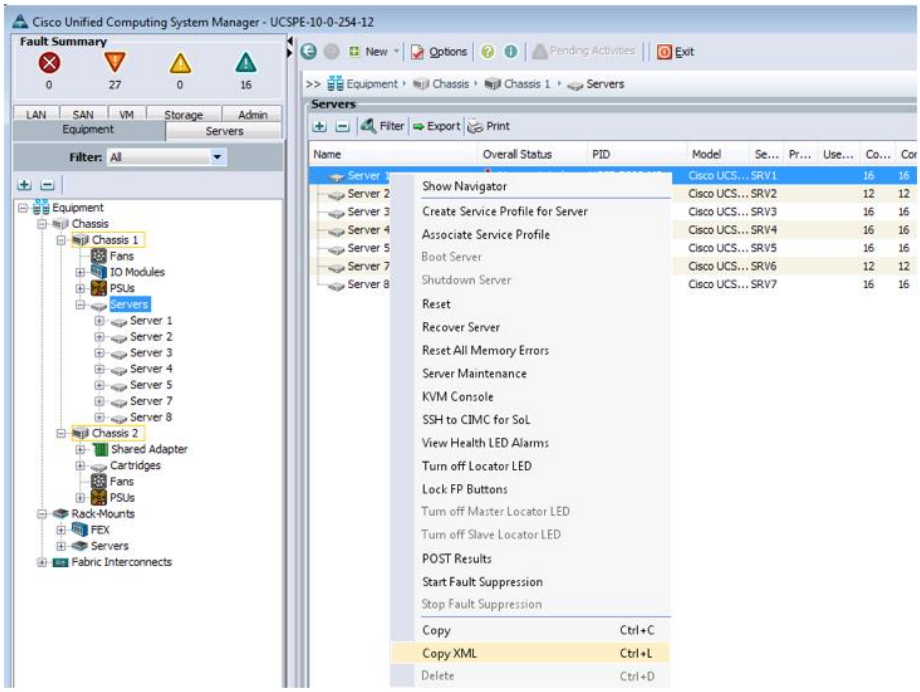
Introduction to UCS Python SDK Part III

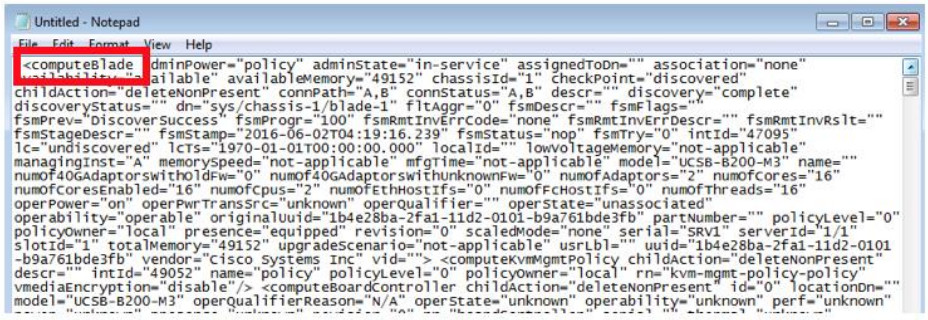
Example #3a: "Copy XML" from the UCSM GUI to to discover XML Class names

Another strategy for getting familiar with the UCS Model, is to discover the XML Class names for many of the objects visible in the GUI.

Next we will discover the XML Class for a UCS Blade, and perform some class queries, save to a variable, and print/parse the output.

In the UCS Manager GUI, you can right click on many objects in the GUI, and select to "Copy XML". On the left pane navigate to the Equipment Tab, Highlight "Servers" (so that the list of servers populates in the right pane). In the right pane, right click on a server, select "Copy XML" to copy to the Windows Clipboard, and then paste to a new Notepad session:





The first reference at the top of the XML is the XML Class name for an object.

- "computeBlade" is the XML Class for a UCS Blade Server

Example #3b: Use Python to query Classes, save to variables, print and parse

Log back into the UCS Domain if necessary.

Execute the following command to query for all instances of the computeBlade XML Class and store in the variable "blades":

```
blades = handle.query_classid("computeBlade")
```

```
>>> blades = handle.query_classid("computeBlade")
>>>
```

We can use the Python function, "len" to determine how many instances of computeBlade were found:

```
len(blades)
```

```
>>> len(blades)
7
>>>
```

There are several methods of printing from this variable to the screen.

- Print the first computeBlade object using **`vars(blades[0])`**
- Note: the numeral in brackets identifies an object (starting with 0)

```
vars(blades[0])
```

```
>>> vars(blades[0])
{'fsm_stage_descr': '', 'dn': 'sys/chassis-1/blade-1', 'num_of_eth_host_ifs': '0
', 'total_memory': '49152', 'scaled_mode': 'none', 'fsm_try': '0', 'num_of_fc_ho
st_ifs': '0', 'sacl': None, 'discovery_status': '', 'fsm_rmt_inv_err_descr': '',
 '_handle': <ucsmsdk.ucshandle.UcsHandle object at 0x032D9190>, 'oper_state': 'u
nassociated', 'presence': 'equipped', 'num_of40_g_adaptors_with_old_fw': '0', 'n
um_of_adaptors': '2', 'lc_ts': '1970-01-01T00:00:00.000', 'num_of_threads': '16'
, 'fsm_rmt_inv_err_code': 'none', 'fsm_prev': 'DiscoverSuccess', '_ManagedObject
__status': None, 'fsm_status': 'nop', 'serial': 'SRV1', 'revision': '0', 'availa
bility': 'available', '_class_id': 'ComputeBlade', 'child_action': None, 'admin_
power': 'policy', 'server_id': '1/1', 'fsm_stamp': '2016-06-02T04:19:16.239', 'a
vailable_memory': '49152', 'descr': '', '_ManagedObject__parent_dn': 'sys/chassi
s-1', 'fsm_progr': '100', 'fsm_flags': '', 'part_number': '', 'chassis_id': '1',
 'check_point': 'discovered', 'policy_owner': 'local', 'local_id': '', 'managing
_inst': 'A', 'oper_qualifier': '', 'mfg_time': 'not-applicable', 'policy_level':
 '0', 'rn': 'blade-1', 'oper_pwr_trans_src': 'unknown', 'discovery': 'complete',
 'status': None, 'vid': '', '_ManagedObject__xtra_props': {}, 'vendor': 'Cisco S
ystems Inc', 'conn_path': 'A,B', 'num_of40_g_adaptors_with_unknown_fw': '0', 'me
mory_speed': 'not-applicable', 'flt_aggr': '0', 'conn_status': 'A,B', 'assigned_
to_dn': '', 'admin_state': 'in-service', 'num_of_cpus': '2', 'int_id': '47095',
'fsm_rmt_inv_rslt': '', '_ManagedObject__parent_mo': None, 'association': 'none'
, 'lc': 'undiscovered', 'name': '', '_child': [], '_dirty_mask': 0, 'upgrade_sce
nario': 'not-applicable', 'low_voltage_memory': 'not-applicable', 'original_uuid
': '1b4e28ba-2fa1-11d2-0101-b9a761bde3fb', 'uuid': '1b4e28ba-2fa1-11d2-0101-b9a7
61bde3fb', '_ManagedObject__xtra_props_dirty_mask': 1, 'fsm_descr': '', 'oper_po
wer': 'on', 'slot_id': '1', 'usr_lbl': '', 'num_of_cores': '16', 'model': 'UCSB-
B200-M3', 'num_of_cores_enabled': '16', 'operability': 'operable'}
>>>
```

You can also print specific properties of this object (or iterate through all the objects to identify certain properties).

Let's view the Model, Serial and Distinguished Name of all objects in the "blades" variable:

```
for blade in blades:
    print blade.model,blade.serial,blade.dn
    <enter>
```

```
>>> for blade in blades:
        print blade.model,blade.serial,blade.dn


UCSB-B200-M3 SRV1 sys/chassis-1/blade-1
UCSB-B200-M4 SRV2 sys/chassis-1/blade-2
UCSB-B200-M3 SRV3 sys/chassis-1/blade-3
UCSB-B200-M3 SRV4 sys/chassis-1/blade-4
UCSB-B420-M3 SRV5 sys/chassis-1/blade-5
UCSB-B200-M4 SRV6 sys/chassis-1/blade-7
UCSB-B200-M3 SRV7 sys/chassis-1/blade-8
>>>
```

There are many properties you can trigger on – feel free to try other combinations (perhaps "total_memory", or "num_of_cpus"

```
for blade in blades:
    print blade.model,total_memory,blade.num_of_cpus
    <enter>
```

```
>>> for blade in blades:
        print blade.total_memory,blade.num_of_cpus


49152 2
49152 2
49152 2
49152 2
49152 4
49152 2
49152 2
>>>
```

Example #3c: Use a Simple Query Filter

Print the blade list again but this time add num_of_cpus to the print command.

```
for blade in blades:
    print blade.model,blade.serial,blade.dn,blade.num_of_cpus
```

```
>>> for blade in blades:
        print blade.model, blade.serial, blade.dn, blade.total_memory, blade.num_of_cpus


UCSB-B200-M4 SRV2 sys/chassis-1/blade-2 49152 2
UCSB-B200-M3 SRV3 sys/chassis-1/blade-3 49152 2
UCSB-B200-M3 SRV4 sys/chassis-1/blade-4 49152 2
UCSB-B420-M3 SRV5 sys/chassis-1/blade-5 49152 4
UCSB-B200-M4 SRV6 sys/chassis-1/blade-7 49152 2
UCSB-B200-M3 SRV7 sys/chassis-1/blade-8 49152 2
UCSB-B200-M3 SRV1 sys/chassis-1/blade-1 49152 2
```

Notice that the blade model UCSB-B420-M3 num_of_cpus value is 4, a filter could be specified as part of the Class ID query to only retrieve blades where the num_of_cpus is equal to 4.

```
blades = handle.query_classid("computeBlade", filter_str="(num_of_cpus, '4',
type='eq')")
for blade in blades:
      print blade.model,blade.serial,blade.dn,blade.num_of_cpus
```

```
>>> blades = handle.query_classid("computeBlade", filter_str="(num_of_cpus, '4', type='eq')")
>>> for blade in blades:
        print blade.model, blade.serial, blade.dn, blade.total_memory, blade.num_of_cpus


UCSB-B420-M3 SRV5 sys/chassis-1/blade-5 49152 4
```

REFERENCE:

Filters as defined by the documentation in the source for the ucshandle module

(See the section: def query_classid in c:\Python27\Lib\site-packages\ucsmsdk\ucshandle.py)

filter_str(str): query objects with specific property with specific value or pattern specifying value.

- (property_name, "property_value, type="filter_type")
  - property_name: Name of the Property
  - property_value: Value of the property (str or regular expression)
  - filter_type:
    - eq - equal to
    - ne - not equal to
    - ge - greater than or equal to
    - gt - greater than
    - le - less than or equal to
    - lt - less than
    - re - regular expression - the default value
  - logical filter type:
    - not
    - and
    - or
  - flag
    - I - case insensitive
    - C - case sensitive - the default value
  - Examples:
    - filter_str="(usr_lbl, 'test', type='eq')"
    - filter_str="(dn,'org-root/ls-C1_B1', type='eq') or (name, 'event', type='re', flag='I')"

Example #3d: Use a Composite Query Filter

This time retrieve blades that have num_of_cpus eq to 2 and are in various numbered slots.

- 2 CPU, and located in slots 1, 3, 5, or 8

```
blades = handle.query_classid("computeBlade", filter_str="(num_of_cpus,
'2', type='eq') and ((slot_id, '1', type='eq') or (slot_id, '3',
type='eq') or (slot_id, '5', type='eq') or (slot_id, '8', type='eq'))")

for blade in blades:
    print blade.model,blade.serial,blade.dn,blade.num_of_cpus
```

```
>>> blades = handle.query_classid("computeBlade", filter_str="(num_of_cpus, '2', type='eq') and ((slot_id, '1', type='eq') or
(slot_id, '3', type='eq') or (slot_id, '5', type='eq') or (slot_id, '8', type='eq'))")
>>> for blade in blades:
        print blade.model, blade.serial, blade.dn, blade.total_memory, blade.num_of_cpus


UCSB-B200-M3 SRV3 sys/chassis-1/blade-3 49152 2
UCSB-B200-M3 SRV7 sys/chassis-1/blade-8 49152 2
UCSB-B200-M3 SRV1 sys/chassis-1/blade-1 49152 2
```

Make sure to understand the parenthesis in the filter_str the "and" is a grouping of the "or" operation

Example #3e: View the Python SDK generated XML code that is getting sent to UCS Manager

All the operations in the Python SDK that interact with the UCS Manager are ultimately creating XML to send to the UCS Manager and then parsing the response.

Step 1: Viewing the XML is simply a setting in the handle.

```
handle.set_dump_xml()
blades = handle.query_classid("computeBlade")
```

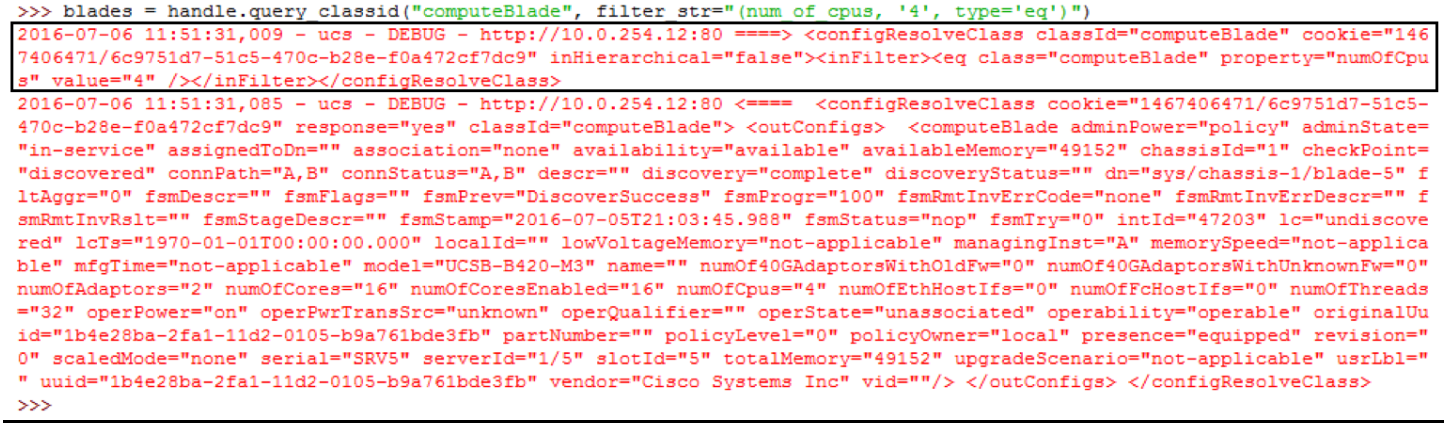
```
>>> handle.set_dump_xml()
>>> blades = handle.query_classid("computeBlade")
2016-07-06 11:45:46,167 - ucs - DEBUG - http://10.0.254.12:80 ====> <configResolveClass classId="computeBlade" cookie="146
7406471/6c9751d7-51c5-470c-b28e-f0a472cf7dc9" inHierarchical="false" />
2016-07-06 11:45:46,242 - ucs - DEBUG - http://10.0.254.12:80 <====  <configResolveClass cookie="1467406471/6c9751d7-51c5-
470c-b28e-f0a472cf7dc9" response="yes" classId="computeBlade"> <outConfigs>  <computeBlade adminPower="policy" adminState=
"in-service" assignedToDn="" association="none" availability="available" availableMemory="49152" chassisId="1" checkPoint=
"discovered" connPath="A,B" connStatus="A,B" descr="" discovery="complete" discoveryStatus="" dn="sys/chassis-1/blade-1" f
ltAggr="0" fsmDescr="" fsmFlags="" fsmPrev="DiscoverSuccess" fsmProgr="100" fsmRmtInvErrCode="none" fsmRmtInvErrDescr="" f
smRmtInvRslt="" fsmStageDescr="" fsmStamp="2016-07-05T21:02:39.018" fsmStatus="nop" fsmTry="0" intId="47107" lc="undiscove
red" lcTs="1970-01-01T00:00:00.000" localId="" lowVoltageMemory="not-applicable" managingInst="A" memorySpeed="not-applica
ble" mfgTime="not-applicable" model="UCSB-B200-M3" name="" numOf40GAdaptorsWithOldFw="0" numOf40GAdaptorsWithUnknownFw="0"
```

From the image above this is the query

```
<configResolveClass
  classId="computeBlade"
  cookie="1464202102/7e490824-b8ef-422f-bb7b-3318a135eba8"
  inHierarchical="false" />
```

Step 2: Now try the simple filter_str query

```
blades = handle.query_classid("computeBlade", filter_str="(num_of_cpus,
'4', type='eq')")
```



From the image above this is the query

```
<configResolveClass
 classId="computeBlade"
 cookie="1464202102/7e490824-b8ef-422f-bb7b-3318a135eba8"
 inHierarchical="false">
 <inFilter><eq class="computeBlade" property="numOfCpus" value="4" /></inFilter>
</configResolveClass>
```

Step 3: Now try the complex filter_str query

```
blades = handle.query_classid("computeBlade", filter_str="(num_of_cpus,
'2', type='eq') and ((slot_id, '1', type='eq') or (slot_id, '3',
type='eq') or (slot_id, '5', type='eq') or (slot_id, '8', type='eq'))")
```

```
>>> blades = handle.query_classid("computeBlade", filter_str="(num_of_cpus, '2', type='eq') and ((slot_id, '1', type='eq')
or (slot_id, '3', type='eq') or (slot_id, '5', type='eq') or (slot_id, '8', type='eq'))")
2016-07-06 11:56:41,865 - ucs - DEBUG - http://10.0.254.12:80 ====> <configResolveClass classId="computeBlade" cookie="146
7406471/6c9751d7-51c5-470c-b28e-f0a472cf7dc9" inHierarchical="false"><inFilter><and><eq class="computeBlade" property="num
OfCpus" value="2" /><or><eq class="computeBlade" property="slotId" value="1" /><eq class="computeBlade" property="slotId"
value="3" /><eq class="computeBlade" property="slotId" value="5" /><eq class="computeBlade" property="slotId" value="8" />
</or></and></inFilter></configResolveClass>
2016-07-06 11:56:41,953 - ucs - DEBUG - http://10.0.254.12:80 <====  <configResolveClass cookie="1467406471/6c9751d7-51c5-
470c-b28e-f0a472cf7dc9" response="yes" classId="computeBlade"> <outConfigs>  <computeBlade adminPower="policy" adminState=
"in-service" assignedToDn="" association="none" availability="available" availableMemory="49152" chassisId="1" checkPoint=
"discovered" connPath="A,B" connStatus="A,B" descr="" discovery="complete" discoveryStatus="" dn="sys/chassis-1/blade-1" f
ltAggr="0" fsmDescr="" fsmFlags="" fsmPrev="DiscoverSuccess" fsmProgr="100" fsmRmtInvErrCode="none" fsmRmtInvErrDescr="" f
smRmtInvRslt="" fsmStageDescr="" fsmStamp="2016-07-05T21:02:39.018" fsmStatus="nop" fsmTry="0" intId="47107" lc="undiscove
red" lcTs="1970-01-01T00:00:00.000" localId="" lowVoltageMemory="not-applicable" managingInst="A" memorySpeed="not-applica
ble" mfgTime="not-applicable" model="UCSB-B200-M3" name="" numOf40GAdaptorsWithOldFw="0" numOf40GAdaptorsWithUnknownFw="0"
numOfAdaptors="2" numOfCores="16" numOfCoresEnabled="16" numOfCpus="2" numOfEthHostIfs="0" numOfFcHostIfs="0" numOfThreads
="16" operPower="on" operPwrTransSrc="unknown" operQualifier="" operState="unassociated" operability="operable" originalUu
id="1b4e28ba-2fa1-11d2-0101-b9a761bde3fb" partNumber="" policyLevel="0" policyOwner="local" presence="equipped" revision="
0" scaledMode="none" serial="SRV1" serverId="1/1" slotId="1" totalMemory="49152" upgradeScenario="not-applicable" usrLbl="
" uuid="1b4e28ba-2fa1-11d2-0101-b9a761bde3fb" vendor="Cisco Systems Inc" vid=""/>  <computeBlade adminPower="policy" admin
State="in-service" assignedToDn="" association="none" availability="available" availableMemory="49152" chassisId="1" check
Point="discovered" connPath="A,B" connStatus="A,B" descr="" discovery="complete" discoveryStatus="" dn="sys/chassis-1/blad
e-3" fltAggr="0" fsmDescr="" fsmFlags="" fsmPrev="DiscoverSuccess" fsmProgr="100" fsmRmtInvErrCode="none" fsmRmtInvErrDesc
r="" fsmRmtInvRslt="" fsmStageDescr="" fsmStamp="2016-07-05T21:02:39.428" fsmStatus="nop" fsmTry="0" intId="47155" lc="und
iscovered" lcTs="1970-01-01T00:00:00.000" localId="" lowVoltageMemory="not-applicable" managingInst="A" memorySpeed="not-a
pplicable" mfgTime="not-applicable" model="UCSB-B200-M3" name="" numOf40GAdaptorsWithOldFw="0" numOf40GAdaptorsWithUnknown
Fw="0" numOfAdaptors="2" numOfCores="16" numOfCoresEnabled="16" numOfCpus="2" numOfEthHostIfs="0" numOfFcHostIfs="0" numOf
Threads="16" operPower="on" operPwrTransSrc="unknown" operQualifier="" operState="unassociated" operability="operable" ori
ginalUuid="1b4e28ba-2fa1-11d2-0103-b9a761bde3fb" partNumber="" policyLevel="0" policyOwner="local" presence="equipped" rev
```

From the image above this is the query

```
<configResolveClass
 classId="computeBlade"
 cookie="1464202102/7e490824-b8ef-422f-bb7b-3318a135eba8"
 inHierarchical="false">
 <inFilter>
  <and><eq class="computeBlade" property="numOfCpus" value="2" />
   <or><eq class="computeBlade" property="slotId" value="1" />
     <eq class="computeBlade" property="slotId" value="3" />
     <eq class="computeBlade" property="slotId" value="5" />
     <eq class="computeBlade" property="slotId" value="8" />
   </or>
  </and>
 </inFilter>
</configResolveClass>
```

Now turn off XML Viewing

```
handle.unset_dump_xml()
```

```
>>> handle.unset_dump_xml()
>>>
```

Example #3f: Other Query Methods

The UCS Manager API supports multiple query Methods

- query_classid
- query_classids
- query_dn
- query_dns
- query_children

Querying multiple classes is as simple as providing a comma separated list of classIds, display the dict keys and the values for the classIds.

```
classes_dict = handle.query_classids("OrgOrg","ComputeBlade")
print classes_dict.keys()
for ucs_item in classes_dict["OrgOrg"]:
  print ucs_item
  <enter>
```

```
>>> print classes_dict.keys()
['OrgOrg', 'ComputeBlade']
>>> for ucs_item in classes_dict["OrgOrg"]:
        print ucs_item


Managed Object                  :       OrgOrg
--------------
child_action                    :None
descr                           :
dn                              :org-root
flt_aggr                        :0
level                           :root
name                            :root
perm_access                     :yes
rn                              :org-root
sacl                            :None
status                          :None


>>>
```

Try the same code for the "ComputeBlade" class Id

```
for ucs_item in classes_dict["ComputeBlade"]:
   print ucs_item
   <enter>
```

```
>>> for ucs_item in classes_dict["ComputeBlade"]:
        print ucs_item


Managed Object                  :       ComputeBlade
--------------
admin_power                     :policy
admin_state                     :in-service
assigned_to_dn                  :
association                     :none
availability                    :available
available_memory                :49152
chassis_id                      :1
check_point                     :discovered
child_action                    :None
conn_path                       :A,B
conn_status                     :A,B
descr                           :
discovery                       :complete
discovery_status                :
dn                              :sys/chassis-1/blade-1
flt_aggr                        :0
fsm_descr                       :
fsm_flags                       :
fsm_prev                        :DiscoverSuccess
fsm_progr                       :100
fsm_rmt_inv_err_code            :none
fsm_rmt_inv_err_descr           :
fsm_rmt_inv_rslt                :
fsm_stage_descr                 :
fsm_stamp                       :2016-07-05T21:51:18.096
fsm_status                      :nop
fsm_try                         :0
int_id                          :47095
lc                              :undiscovered
lc_ts                           :1970-01-01T00:00:00.000
local_id                        :
low_voltage_memory              :not-applicable
managing_inst                   :A
memory_speed                    :not-applicable
```

To query a specific dn or distinguished name of a UCS Object use the query_dn function.
Note: The dn of an object is an unambiguous reference to the object in the entire UCS object model.

blade_by_dn = handle.query_dn("sys/chassis-1/blade-1")
print blade_by_dn.dn
print blade_by_dn.dn,blade_by_dn.num_of_cpus,blade_by_dn.serial

```
>>> blade_by_dn = handle.query_dn("sys/chassis-1/blade-1")
>>> print blade_by_dn.dn
sys/chassis-1/blade-1
>>> print blade_by_dn.dn,blade_by_dn.num_of_cpus,blade_by_dn.serial
sys/chassis-1/blade-1 2 SRV1
>>>
```

To query multiple dns use the query_dns function.

blades_dict = handle.query_dns("sys/chassis-1/blade-1","sys/chassis-1/blade-5")
print blades_dict.keys()
print blades_dict['sys/chassis-1/blade-5'].model
print blades_dict['sys/chassis-1/blade-1'].model

```
>>> blades_dict = handle.query_dns("sys/chassis-1/blade-1","sys/chassis-1/blade-5")
>>> print blades_dict.keys()
['sys/chassis-1/blade-5', 'sys/chassis-1/blade-1']
>>> print blades_dict['sys/chassis-1/blade-5'].model
UCSB-B420-M3
>>> print blades_dict['sys/chassis-1/blade-1'].model
UCSB-B200-M3
>>>
```

To query children objects use the query_children function. The query_children function has the ability to query all the children objects of an objects or limit the query to only a particular class id of the children objects.

```
blade_by_dn = handle.query_dn("sys/chassis-1/blade-1")
blade_by_dn_children = handle.query_children(in_mo=blade_by_dn)
for blade_child_object in blade_by_dn_children:
   print blade_child_object.dn
        <enter>
```

```
>>> blade_by_dn = handle.query_dn("sys/chassis-1/blade-1")
>>> blade_by_dn_children = handle.query_children(in_mo=blade_by_dn)
>>> for blade_child_object in blade_by_dn_children:
        print blade_child_object.dn


sys/chassis-1/blade-1/kvm-mgmt-policy-policy
sys/chassis-1/blade-1/boardController
sys/chassis-1/blade-1/memmory-config
sys/chassis-1/blade-1/scrub-policy
sys/chassis-1/blade-1/locator-led
sys/chassis-1/blade-1/fw-status
sys/chassis-1/blade-1/fwsyncack
sys/chassis-1/blade-1/adaptor-2
sys/chassis-1/blade-1/adaptor-1
sys/chassis-1/blade-1/fabric-B
sys/chassis-1/blade-1/budget
sys/chassis-1/blade-1/pnuos
sys/chassis-1/blade-1/board
sys/chassis-1/blade-1/mgmt
sys/chassis-1/blade-1/bios
sys/chassis-1/blade-1/diag
sys/chassis-1/blade-1/fsm
```

Notice that there are two adaptors that are children to the computeBlade, the query_children function can limit the retuned objects to the specific class for adaptors.

```
blade_by_dn = handle.query_dn("sys/chassis-1/blade-1")
blade_by_dn_children = handle.query_children(in_mo=blade_by_dn,class_id="adaptorUnit")
for blade_child_object in blade_by_dn_children:
   print blade_child_object.dn
   <enter>
```

```
>>> blade_by_dn = handle.query_dn("sys/chassis-1/blade-1")
>>> blade_by_dn_children = handle.query_children(in_mo=blade_by_dn,class_id="adaptorUnit
")
>>> for blade_child_object in blade_by_dn_children:
        print blade_child_object.dn


sys/chassis-1/blade-1/adaptor-2
sys/chassis-1/blade-1/adaptor-1
>>>
```

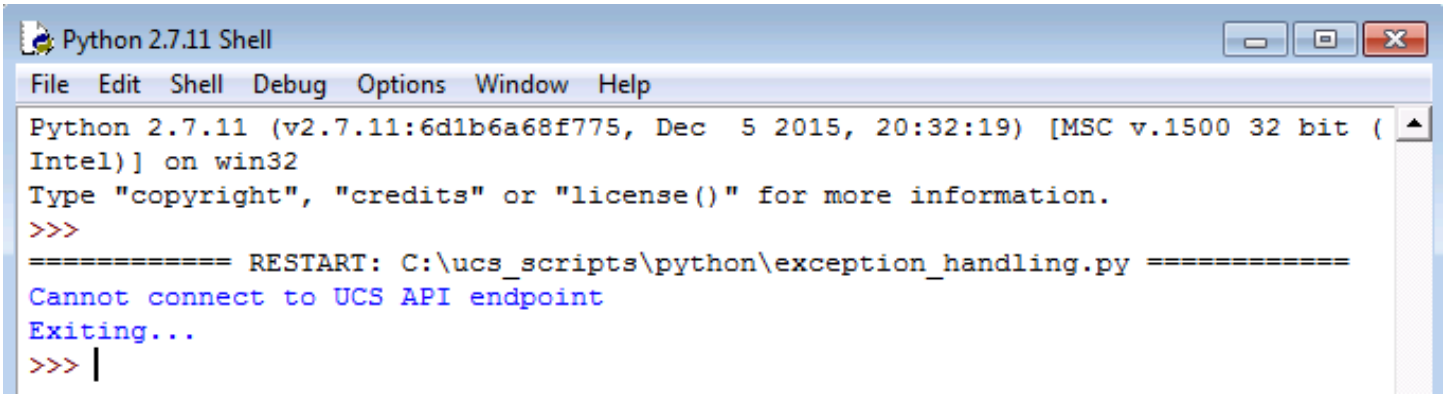
APPENDIX A: Exception Handling

Review C:\ucs_scripts\python\exception_handling.py to see an example of Exception Handling:

```
# Import sys (a module containing system specific parameters and functions)
# Import urllib2 (a module for opening URLs)
# Import the "UcsHandle" class from the "ucshandle" module in the ucsmsdk directory
# Import the "UcsException" class from the "ucssession" module in the ucsmsdk directory
import sys
import urllib2
from ucsmsdk.ucshandle import UcsHandle
from ucsmsdk.ucssession import UcsException


# Initialize a handle -  specify an incorrect IP to demonstrate exception handling
handle = UcsHandle("11.0.254.12","admin","passsword",secure=False)

# Attempt Login - incorporate exception handling
try:
    handle.login()
except urllib2.URLError as eURLError:
    print "Cannot connect to UCS API endpoint"
    print "Exiting..."
    sys.exit(1)
except UcsException as eUcsException:
    print eUcsException
    print "Unable to login"
    print "Exiting..."
    sys.exit(1)
```

Right click on this file, click Edit with IDLE, and then Click Run (Run Module)

- After a few moments, notice that an exeception is encountered when the URL cannot be reached:



APPENDIX B: UCS Python SDK – Automated Provisioning Example

Configuration and Provisioning Overview

In the following examples, the Python SDK is used to automate UCS server configuration following portions of the "FlexPod Datacenter with Red Hat Enterprise Linux OpenStack Platform Design Guide". The complete Cisco Validated Design (CVD) guide and additional information can be downloaded from the following link:
http://www.cisco.com/c/en/us/td/docs/unified_computing/ucs/UCS_CVDs/flexpod_openstack_osp6_design.html

The scripts below automate several steps from the "Cisco UCS and Server Configuration" section of the design guide. Subsections of the design guide are presented below along with corresponding Python SDK scripts to perform the described configuration. At a high-level, the following covers creation of resource pools, policies, and service profile templates for OpenStack server nodes. As described in the design guide, there are three different server "roles" that are configured for the topology: Installer, Controller, and Compute. Multiple roles allow for targeted deployment and scalability, but configuration does require numerous steps that are very similar. Fortunately, many of the steps can be automated to reduce configuration errors and deployment time. Note that in the interest of time for this exercise, not all steps from the CVD are performed. Also note that the intent of this example is to validate the automation process against UCS Platform Emulator. The Platform Emulator provides an excellent platform for verifying the scripts and configuration before deploying against real HW.

Deployment Part I: UUID and Server Pools/Policies

Steps in the design guide can be performed with the UCSM GUI and Python scripts created with convert_to_ucs_python() as shown earlier in the student guide. The scripts presented below use data files for part of the configurations so that configuration code can be separated from data for a given configuration. Each of the scripts has a similar login portion that uses a basic python script to specific admin IP and login credentials.

adminSettings.py file

Locate c:\ucs_scripts\python\adminSettings.py in Windows File Manager, right click and edit in the IDLE Editor:

Login information (ip address, username, etc.) is taken from the adminSettings.py file to avoid specifying in each subsequent script:

```
# admin settings (ips, creds, etc.)

# VM UCSPE 3.1(1e) - local
ip = "172.16.143.186"
user = "ucspe"
pw = "ucspe"
secure = True
```

Skeleton Script

Each configuration script begins by pulling login information from adminSettings.py (or a user specified alternative):

```
import sys
from ucsmsdk import ucshandle
# admin settings are imported from args (or default) below

if __name__ == "__main__":
    try:
        if len(sys.argv) > 2:
            print "Usage: " + sys.argv[0] + " [admin file]"
            print "       [admin file]:   optional admin (IP/user/password) file"
            print "                       adminSettings.py used by default"
            sys.exit(0)
        if len(sys.argv) == 2:
            settingsFile = sys.argv[1]
        else:
            settingsFile = "adminSettings.py"
        settingsFile = settingsFile.split(".")[0]
        print "%s: Using admin settings (IP/user/password) from %s" % (sys.argv[0],
settingsFile)
        settings = __import__(settingsFile)
        handle = ucshandle.UcsHandle(settings.ip, settings.user, settings.pw,
secure=settings.secure)
        handle.login()
```

With login complete and a handle available, scripts are written to carry out the configuration steps.

Create UUID Suffix Pools

For UUID pool creation, the “Create UUID Suffix Pools” section in the design guide describes the following actions to perform in the UCSM GUI

“To configure the necessary universally unique identifier (UUID) suffix pool for the Cisco UCS environment, complete the following steps:

. In Cisco UCS Manager, click the Servers tab in the navigation pane.
. Select Pools > root.
. Right-click UUID Suffix Pools.
. Select Create UUID Suffix Pool.
. Enter UUID_Pool as the name of the UUID suffix pool
. …”

The PythonSDK’s convert_to_ucs_python() can be used when performing the above (and any other actions specified in the design guide) to create scripts. Here’s the example captured script:

```
        ##### Start-Of-PythonScript #####

        from ucsmsdk.mometa.uuidpool.UuidpoolPool import UuidpoolPool
        from ucsmsdk.mometa.uuidpool.UuidpoolBlock import UuidpoolBlock
```

```
        mo = UuidpoolPool(parent_mo_or_dn="org-root", policy_owner="local",
prefix="derived", descr="", assignment_order="default", name="UUID_Pool")
        mo_1 = UuidpoolBlock(parent_mo_or_dn=mo, to="0000-000000000020", r_from="0000-
000000000001")
        handle.add_mo(mo)

        handle.commit()
        ##### End-Of-PythonScript #####
```

createServerPoolsAndPolicies.py script

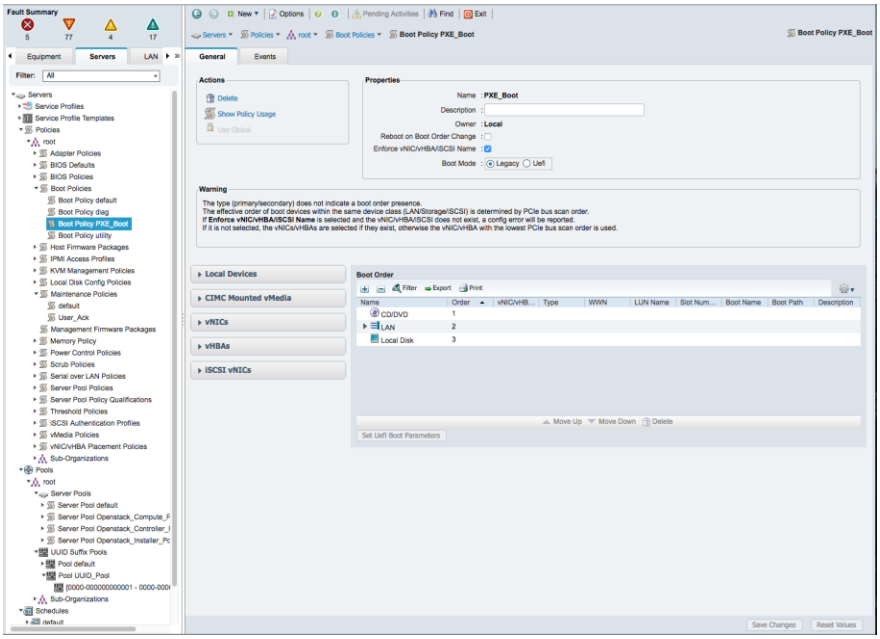
The createServerPoolsAndPolicies.py script builds on the above login and UUID pool creation to also perform the following:

- Create Controller, Compute, and Installer host server pools
- Create vNIC placement policies
- Create Maintenance policies
- Create Boot policies

Run the script

```
python createServerPoolsAndPolicies.py
```

When the script is run against the Platform Emulator, you should see the policies and pools configured. Note that in the example script, iSCSI boot policies are not configured and instead a “PXE_Boot” policy is created for PXE and local disk boot:



Deployment Part II: LAN Pools and Policies

When configuring VLANs, vNICs, and other network related policies, the design guide contains many steps that are similar and that vary only in data used for configuration. To avoid manually performing all

the configuration in the GUI, the following simple example uses configuration data that is kept in Comma Separated Values (CSV) files. The CSV files can be edited with Excel or similar tools, and they are subsequently used to configure UCSM. Python's csv module is used to read configuration parameters and the SDK is used to carry out desired operations.

While this example has limited content in the csv files, Python provides a variety of similar modules for interacting with data management systems. One example use case for the UCSM Python SDK would be interfacing with a CMDB to provide inventory data and help provision servers when needed.

CSV file structure for MAC pools, VLANs, and vNIC configuration

The csv file provides configuration data for the following steps from the design guide:

- Create MAC Address Pools
- Create VLANs
- Create vNIC Templates
    - Including LAN connectivity policies with explicit vNIC placement

The 1st row of each column in the csv file specifies which parameter is being set. Subsequent rows within each column specify values for that parameter. Here's an example for MAC pool values:

MAC_pool	MAC_From	MAC_To
MAC_Pool_A	00:25:B5:00:0A:00	00:25:B5:00:0A:FF
MAC_Pool_B	00:25:B5:00:0B:00	00:25:B5:00:0B:FF

When readying the csv in Python, 'MAC_pool', 'MAC_From', or 'MAC_To' can be used to specify a column to pull data from. In the example scripts, the csv file is read row by row and configuration data is set within UCSM from each row.

createLANPoolsAndPolicies.py script

Python's csv module is used to read configuration settings row by row from the csv file and pass parameters to the various Python SDK configuration functions. Below shows how MAC pools are created using the csv file data:

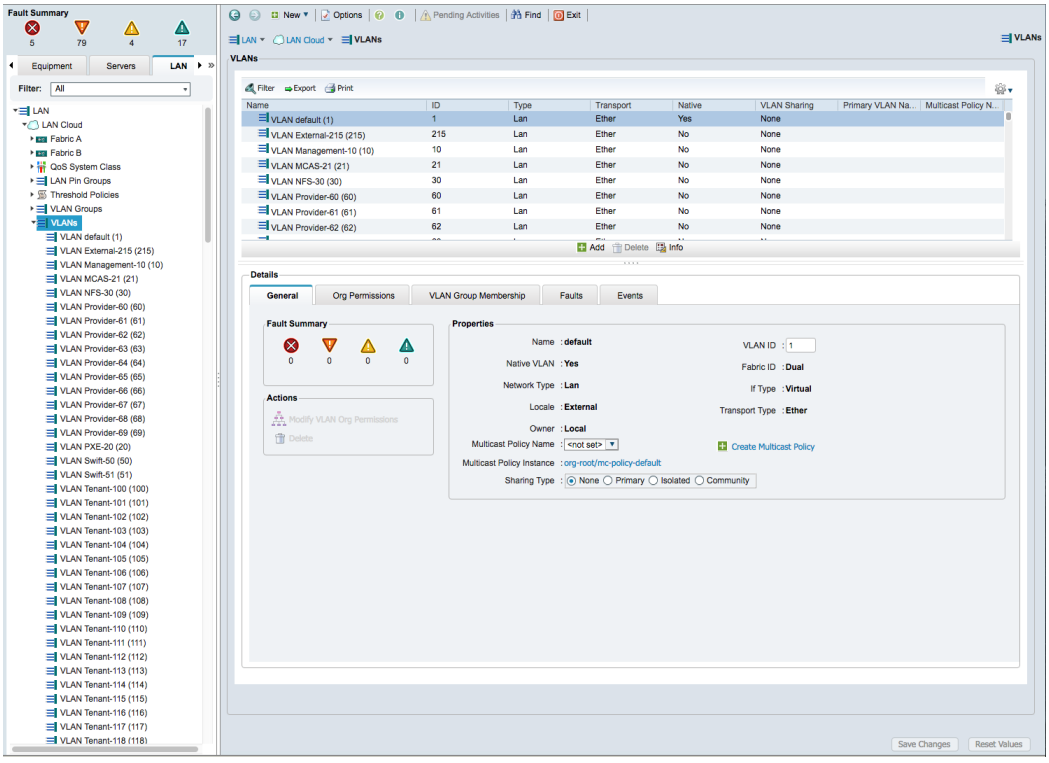
```
# apply settings from each row in the csv file
csvFilename = sys.argv[1]
csvFile = open(csvFilename, "r")
if not csvFile:
    print "Error: could not open %s" % csvFilename
    sys.exit(1)
csvReader = csv.DictReader(csvFile)
rowNum = 1
for row in csvReader:
    rowNum += 1

    # set org (org-root as default)
```

```
            if row['org']:
                org = row['org']
            else:
                org = 'org-root'
            if row['MAC_pool'] and row['MAC_From'] and row['MAC_To']:
                # create MAC pool
                from ucsmsdk.mometa.macpool.MacpoolPool import MacpoolPool
                from ucsmsdk.mometa.macpool.MacpoolBlock import MacpoolBlock

                mo = MacpoolPool(parent_mo_or_dn=org, policy_owner="local", descr="",
assignment_order="default", name=row['MAC_pool'])
                mo_1 = MacpoolBlock(parent_mo_or_dn=mo, to=row['MAC_To'],
r_from=row['MAC_From'])
                handle.add_mo(mo)

                handle.commit()
```

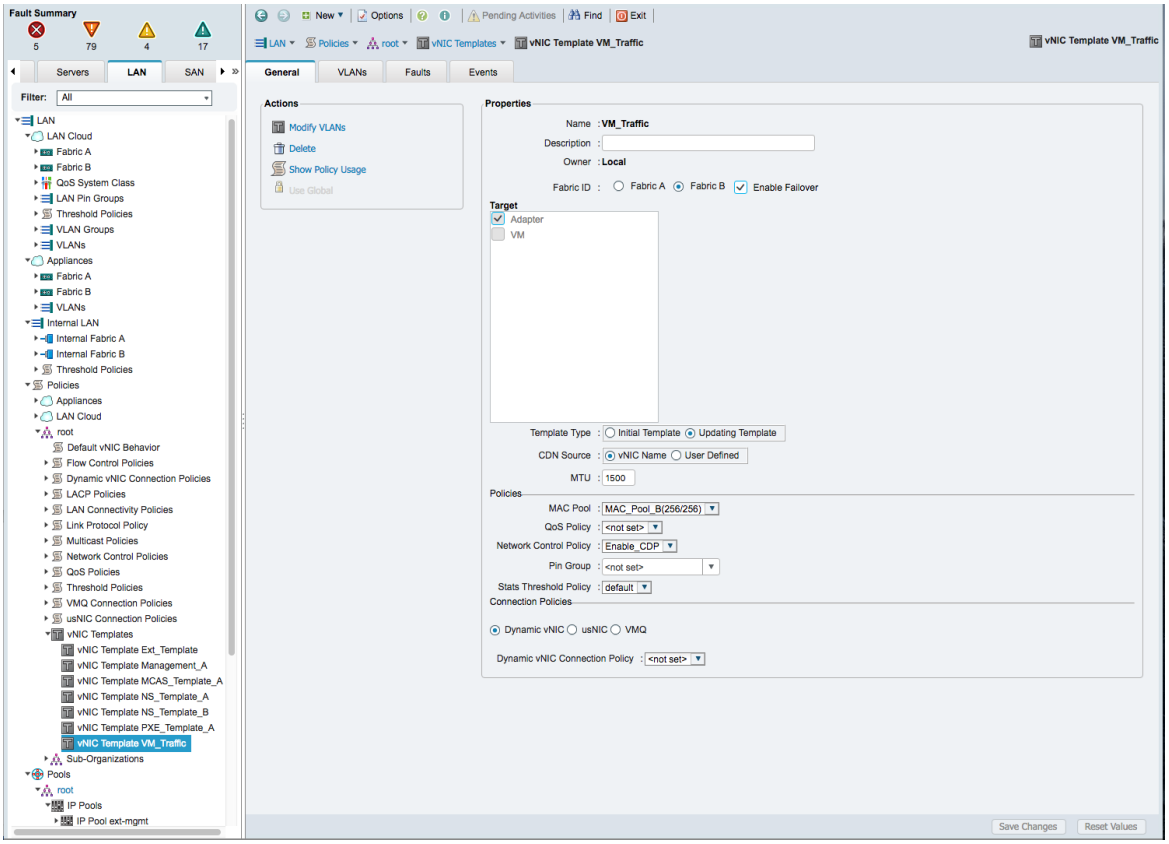
Run the script

```
python createServerPoolsAndPolicies.py
```

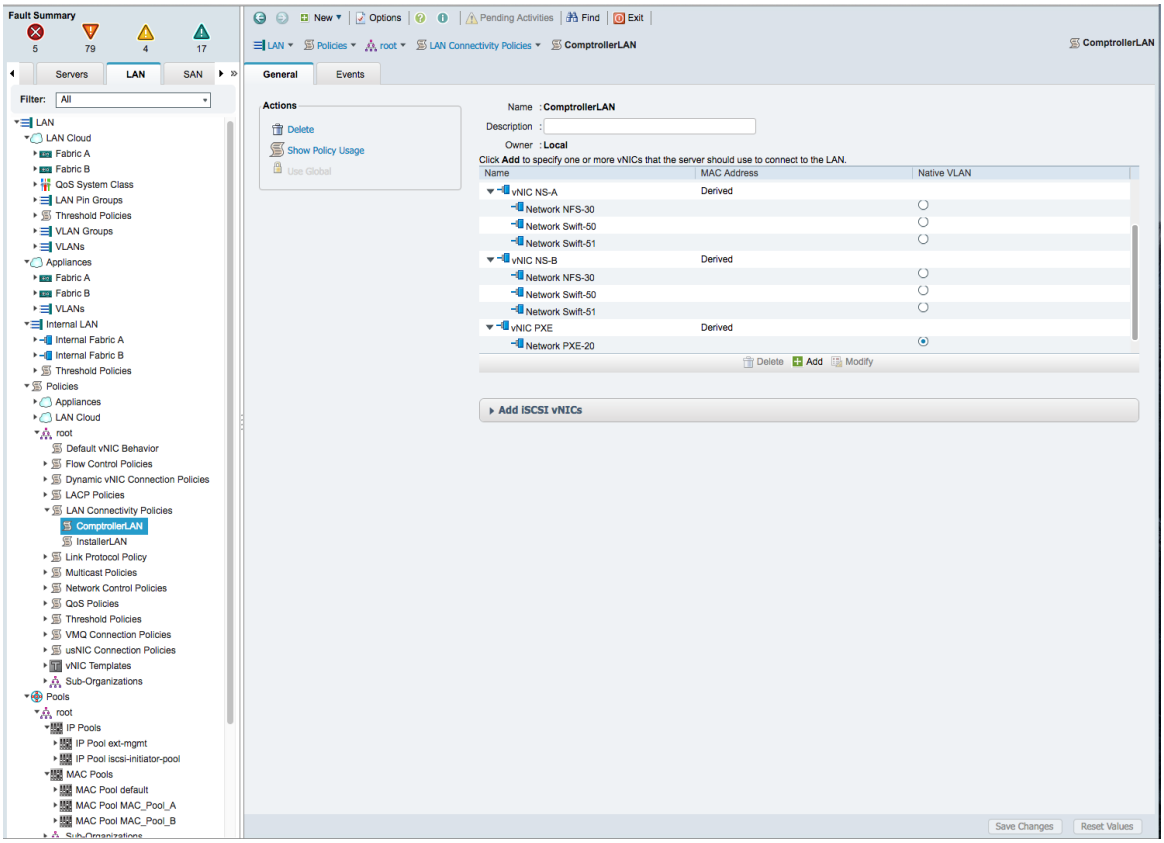
VLAN and vNIC configuration involves many additional settings as shown in the csv and .py script. Once the script is run against the Platform emulator, you should see a large set of VLANs created:



Along with many vNIC templates:



And LAN Connectivity policies that also provide vNIC placement used during subsequent service profile creation:



Deployment Part III: Service Profile Templates and Service Profiles

Similar to VLAN, vNIC, and LAN Connectivity settings that were taken from a csv configuration file, Service Profile Template settings are taken from the ServiceProfilesOpenStack.csv file. The csv file specifies template names, service profile instance names and counts, and policies that are set for the template. The csv file and configuration script follows the design guide steps for the following:

- Create Service Profile Templates
- Create Service Profiles from each template
- Assign profiles to specific server pools

ServiceProfilesOpenStack.csv

The service profiles script is similar to the LAN script and processes each row in the csv file to determine what settings to apply. The profiles script uses the Python SDK samples modules for profile creation, and varies slightly from the design guide in processing. Specifically, in the script templates are created, then profiles are created from the template, then servers are associated with the profile (whereas the design guide specifies pools during template creation). Here's the main portion of the script:

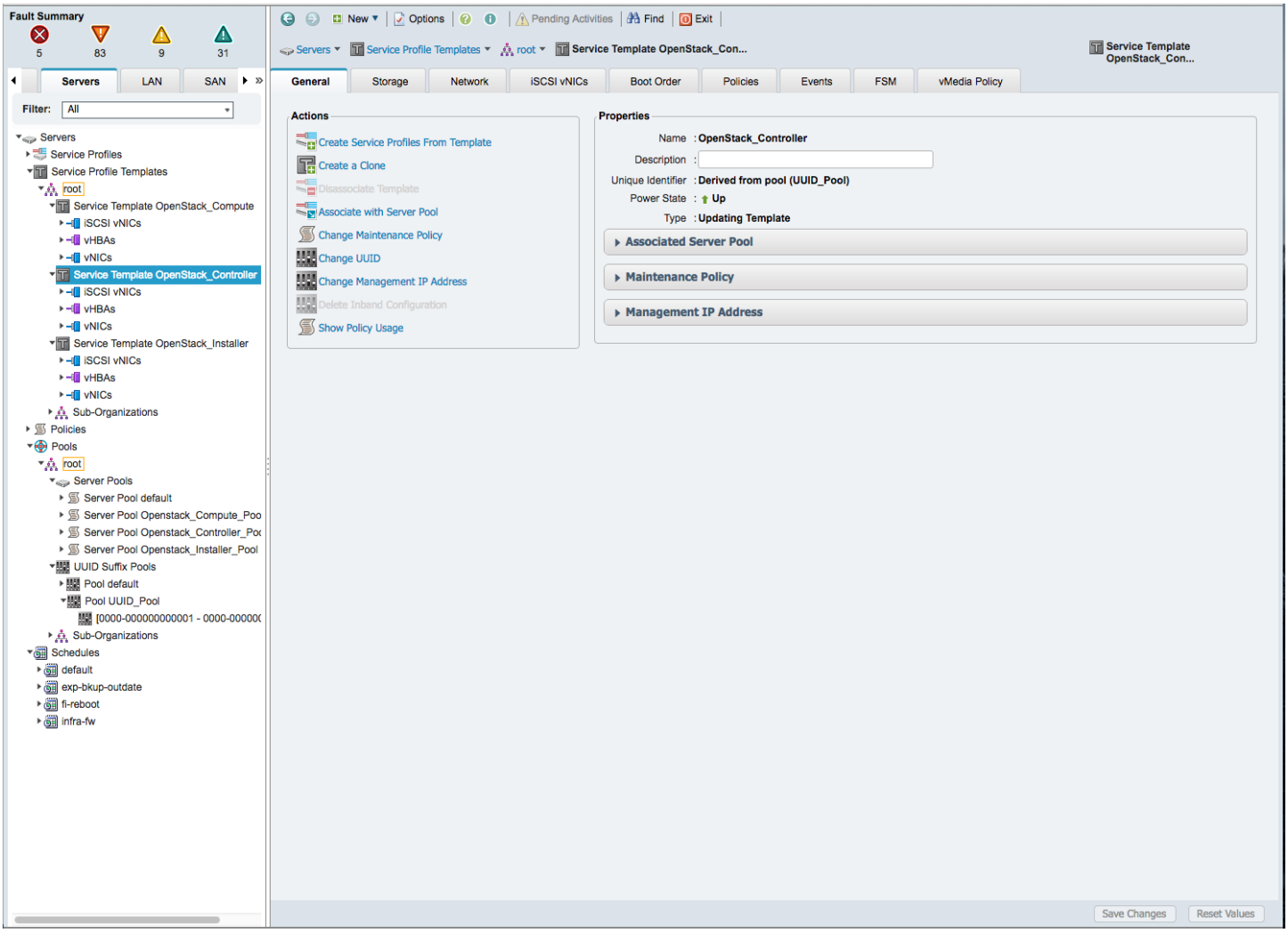
```
        for row in csvReader:
<snip>
            # use the sample template creation function
            mo = service_profile.sp_template_create(handle, name=template,
type="updating-template",
                                                    resolve_remote="yes", descr="",
usr_lbl="",
                                                    src_templ_name="", ext_ip_state="pooled",
                                                    ext_ip_pool_name=row['Mgmt_IP_Pool'],
```

```
                                        ident_pool_name=row['UUID'],
                                        agent_policy_name="",
                                        bios_profile_name="",
                                        boot_policy_name=row['Boot_Policy'],
                                        dynamic_con_policy_name="",
                                        host_fw_policy_name="",
                                        kvm_mgmt_policy_name="",
                                        lan_conn_policy_name=row['LAN_Policy'],
                                        local_disk_policy_name="",
                                        maint_policy_name=row['Maint_Policy'],
                                        mgmt_access_policy_name="",
                                        mgmt_fw_policy_name="",
                                        power_policy_name="",
                                        san_conn_policy_name="",
                                        scrub_policy_name="",
                                        sol_policy_name="",
                                        stats_policy_name="",
                                        vcon_profile_name=row['vNIC_placement'],
                                        vmedia_policy_name="",
                                        parent_dn=org)
        # create the actual service profiles
        service_profile.sp_create_from_template(handle,
                                                naming_prefix=profileName,
                                                name_suffix_starting_number="1",
                                                number_of_instance=numInstances,
                                                sp_template_name=template,
                                                in_error_on_existing="true",
                                                parent_dn=org)
        serverPool = row['Server-Pool'] if (row['Server-Pool']) else 'default'
        for instance in range(1, numInstances + 1):
            dn = "%s/ls-%s%d" % (org, profileName, instance)
            mo = LsRequirement(parent_mo_or_dn=dn, restrict_migration="no",
                               name=serverPool, qualifier="")
            handle.add_mo(mo, True)
            handle.commit()
```

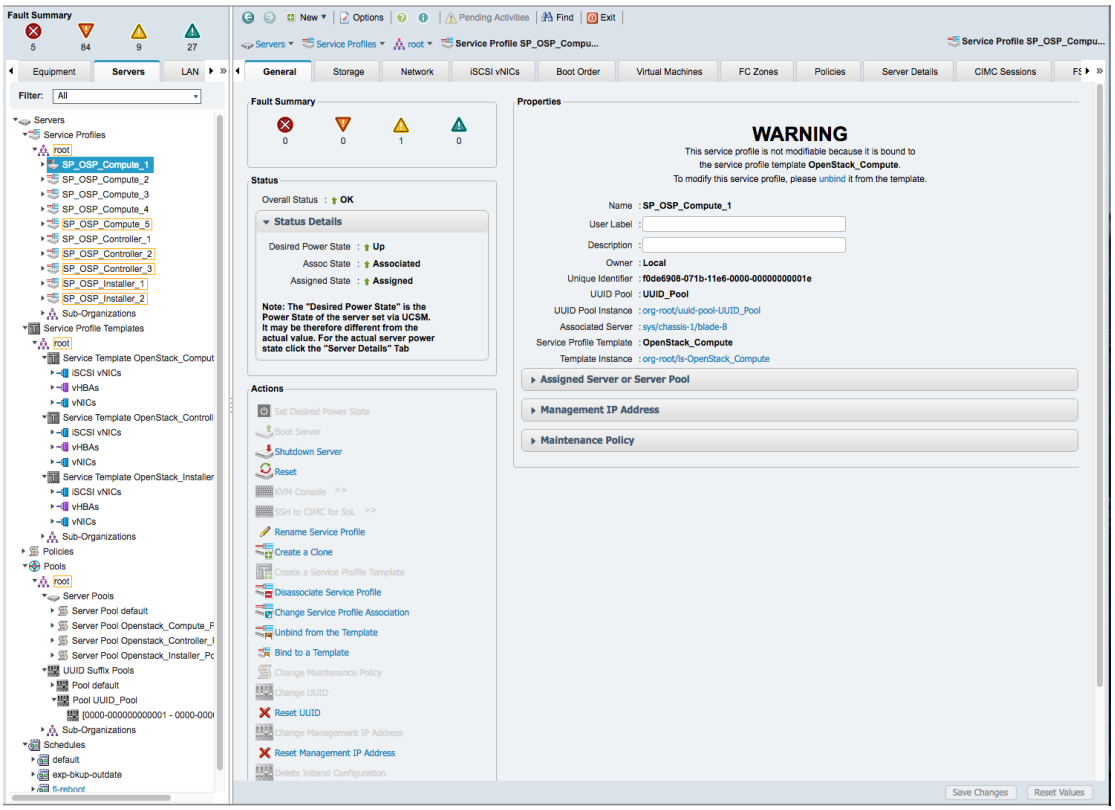
Run the script

```
python createServiceProfiles.py ServiceProfilesOpenStack.csv
```

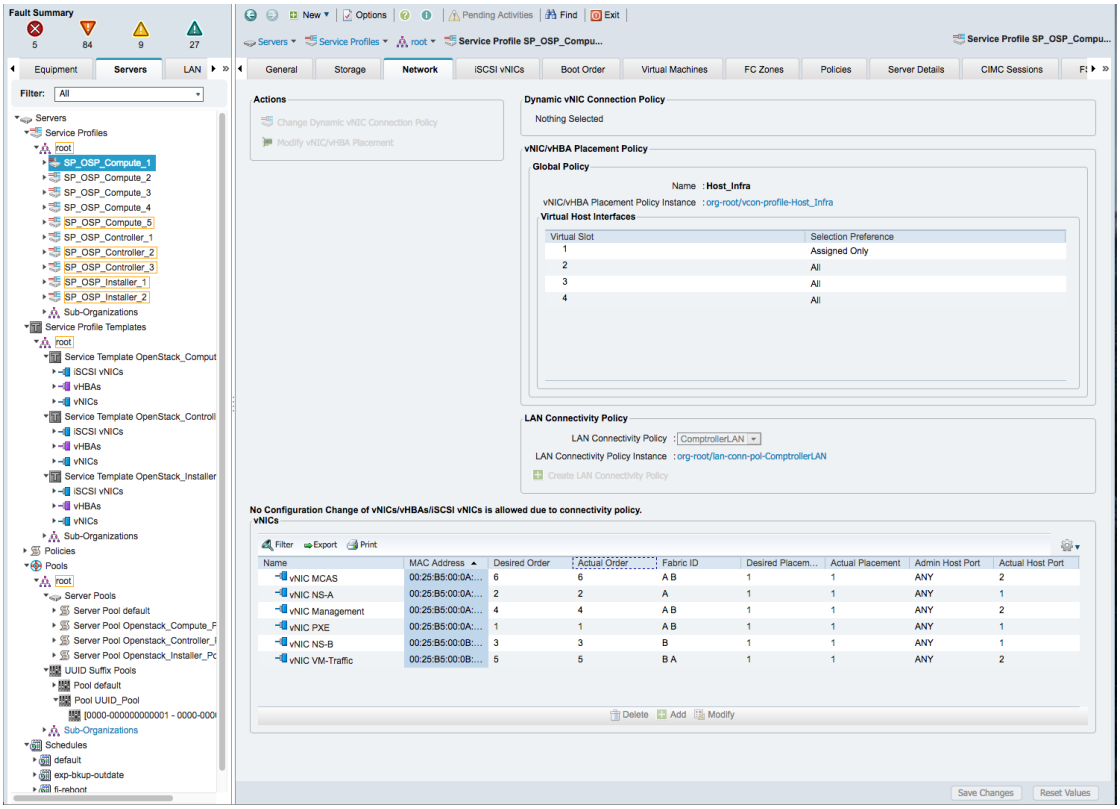
Once the script is run, you should see Service Profile Templates for each OpenStack node type:



Profiles should also be created and associated with servers. After a few minutes, the Platform Emulator should show each profile as configured:



You can also verify vNIC placement (as requested in the design guide). Note that the LAN Connectivity policies created previously actually specified placement order, and when the profiles where created the order is taken from the LAN policy:



Deployment Part IV: iSCSI Boot and other policy creation

You may have noticed that the previous deployment steps do not cover everything in the design guide. The following are some of the missing configuration steps:

- Firmware management policies
- Uplink configuration
- iSCSI boot policies (including target setup)

The example scripts can be expanded to include the above if desired and the Platform Emulator can be used to verify the updated scripts.

Congratulations, you have completed the UCS Python SDK for UCS Manager Lab!