Converting a Cloudify Plugin to Support Both Python2 and Python3

Background

Cloudify currently is built on top of Python 2.7. Formal support of Python2 has been scheduled (for the past 5 years) to go away on 1/1/2020.

Cloudify will be coming out with a version of the cloudify manager that runs on Python3 in 4Q2019. As part of this, they will need to convert all of their plugins to also run on Python3.

Unfortunately, any plugins that we have also need to be converted to Python3 in order to run on the new platform.

The Conversion Process

To prepare for this, we need to also convert our plugins to run under Python3. However, because of the timing we also need the code to continue running under Python2. To do this, we will follow these steps:

1. make sure tests run under Python2
2. convert code to be compatible with both Python2 and Python3
3. make sure tests continue to run under Python2 AND Python3

Make Sure that Python2 Tests Work

Before any conversion is done, make certain that you have tests for your code.

Many of the plugins are already set up with tox-based tests; these MUST be able to be run.

If tox-based tests do NOT exist, they should be created or an alternative form of testing platform needs to be provided.

The key point is: you need to have tests that can be run on your plugin code that is SEPARATE from it being run on a cloudify manager. That is, make certain that "`tox test`" runs cleanly for Python2.

Conversion to Both Python2 and Python3

There are a number of ways that this step can be accomplished. This is one way that seems to work fine:

Step 1: Create a Python3 Version of Your Files

Python3 comes with the `2to3` command. Use the `2to3` command with the `-w` option to create a Python3 version of each file:

```
$ 2to3 -w *.py
```

For each file where differences were found that needed to be made, you will now find both a .bak file and an updated file. If there are no

```
$ ls -l *.py*
-rw-r--r-- 1 th1395 th1395  3965 Sep 14  2018 ecomp_logger.py
-rw-rw-r-- 1 th1395 th1395   490 Jul 23 19:34 __init__.py
-rw-rw-r-- 1 th1395 th1395   483 Jul 23 19:33 __init__.py.bak
-rw-rw-r-- 1 th1395 th1395  2597 Jul 23 19:34 logginginterface.py
-rw-rw-r-- 1 th1395 th1395  2590 Jul 23 19:33 logginginterface.py.bak
-rw-rw-r-- 1 th1395 th1395 28763 Jul 23 19:34 pgaas_plugin.py
-rw-rw-r-- 1 th1395 th1395 28738 Jul 23 19:33 pgaas_plugin.py.bak
```

Now rename each of your files that had differences to use "`.py3`" extensions:

```
$ for i in *.py;do if [ -f $i.bak ]; then mv $i ${i}3;fi;done
```

Step 2: Merge the Two Versions

For each `.py3` file, you will now create a file that has BOTH the Python2 AND the Python3 code present, with the differences clearly marked. The Linux GNU diff command has a nice facility to do this:

```
$ for i in *.py3;do b=$(basename $i .py3); diff -DUSING_PYTHON2 $b.py.bak $b.py3 > $b.py;done
```

Your `.py` files will now have a series of blocks that look like this, using C preprocessor syntax:

NOTE: The sense of the "#ifdef/#ifndef" is backwards from what we really want,

but this will be dealt with in a later section below.

I've added annotated the lines further to make it clearer which lines are the python2 and which are the python3 code.

```
. . .

#ifndef USING_PYTHON2

from logginginterface import *                           # python2 code

#else /* USING_PYTHON2 */

from .logginginterface import *                          # python3 code

#endif /* USING_PYTHON2 */

. . .

#ifndef USING_PYTHON2

import urllib                                            # python2 code

#else /* USING_PYTHON2 */

import urllib.request, urllib.parse, urllib.error        # python3 code

#endif /* USING_PYTHON2 */

. . .

#ifndef USING_PYTHON2

  return urllib.quote(str(s), '')                        # python2 code

#else /* USING_PYTHON2 */

  return urllib.parse.quote(str(s), '')                  # python3 code

#endif /* USING_PYTHON2 */

. . .

#ifndef USING_PYTHON2

  os.umask(077)                                         # python2 code

#else /* USING_PYTHON2 */

  os.umask(0o77)                                        # python3 code

#endif /* USING_PYTHON2 */
```

```
. . .
#ifndef USING_PYTHON2
  if openstack.has_key('custom_configuration'):           # python2 code
#else /* USING_PYTHON2 */
  if 'custom_configuration' in openstack:                 # python3 code
#endif /* USING_PYTHON2 */
. . .
#ifndef USING_PYTHON2
  print >>fp, binascii.hexlify(b).decode('utf-8')        # python2 code
#else /* USING_PYTHON2 */
  print(binascii.hexlify(b).decode('utf-8'), file=fp)    # python3 code
#endif /* USING_PYTHON2 */
. . .
#ifndef USING_PYTHON2
  debug("ctx node has the following Properties: {}".format(i.properties.keys()))
#else /* USING_PYTHON2 */
  debug("ctx node has the following Properties: {}".format(list(i.properties.keys())))
#endif /* USING_PYTHON2 */
. . .
#ifndef USING_PYTHON2
  for k, nv in want.items():                           # python2 code
#else /* USING_PYTHON2 */
  for k, nv in list(want.items()):                     # python3 code
#endif /* USING_PYTHON2 */
. . .
```

These .py files are currently executable by NEITHER Python2 NOR Python3, so the next step is to deal with the differences to make them runnable by BOTH.

- There are some things allowed in Python2.7 that are mandated in Python3.
- There are other things that can be done in just the Python3 form if the right settings are provided.
- Or there are things that can be done in just the Python2 form.
- There are things that must be done differently in the two languages, so a run-time choice is needed.

We will make use of each of these features.

Preparatory Statements

First step is to add these statements to the very top of each .py file:

```
from __future__ import print_function
import sys
USING_PYTHON2 = sys.version_info[0] < 3
```

If your `.py` file starts with a `#!` invocation statement, place those lines AFTER the shbang:

```
#!/usr/bin/env python
from __future__ import print_function
import sys
USING_PYTHON2 = sys.version_info[0] < 3
```

These lines do two things:

1. Allows the function version of the print statements to be used.
2. Gives us a value (`USING_PYTHON2`) that can be tested for at run time to determine if we are running Python2 or Python3.

Changing the #ifndef/#else/#endif Statements

You need to change blocks such as this:

```
#ifndef USING_PYTHON2

some python2 code

#else /* USING_PYTHON2 */

some python3 code

#endif /* USING_PYTHON2 */


#ifdef USING_PYTHON2

   some python3 code

#endif /* USING_PYTHON2 */
```

to this. Make CERTAIN that the indentation is adjusted as well.

```
if USING_PYTHON2:

  some python2 code

else:

  some python3 code


if not USING_PYTHON2:

  some python3 code
```

Yes, there is some cognitive dissonance for changing "#ifndef" to "if", but the rest of the generated code is set up properly to be able to quickly edit the code.

- Search for all instances of "`#ifndef USING_PYTHON2`" and change "`#ifndef`" to "`if`" and add a trailing "`:`".
- Search for all instances of "`#else /* USING_PYTHON2 */`" and change that to "else:".
- Remove all instances of "`#endif /* USING_PYTHON2 */`" from the code.
- Re-indent the lines in between.

- `You should also search for "#ifdef USING_PYTHON2" and change that to "if not USING_PYTHON2:".`

Import Statements

The way that imports of functions found in local files changes in Python3. So change blocks such as this:

```
#ifndef USING_PYTHON2

from logginginterface import *      # python2 code

#else /* USING_PYTHON2 */

from .logginginterface import *     # python3 code

#endif /* USING_PYTHON2 */
```

to this. Make CERTAIN that the indentation is adjusted as well.

```
if USING_PYTHON2:
  from logginginterface import *
else:
  from .logginginterface import *
```

In some cases, standard library interfaces have changed. So change blocks such as this:

```
#ifndef USING_PYTHON2
import urllib     # python2 code
#else /* USING_PYTHON2 */
import urllib.request, urllib.parse, urllib.error     # python3 code
#endif /* USING_PYTHON2 */
```

to this. Make CERTAIN that the indentation is adjusted as well.

```
if USING_PYTHON2:
  import urllib
else:
  import urllib.request, urllib.parse, urllib.error
```

Multiple such import statements can be combined together, as in:

```
if USING_PYTHON2:
  from logginginterface import *     # python2 code
  import urllib
else:
  from .logginginterface import *     # python3 code
  import urllib.request, urllib.parse, urllib.error
```

Standard Library Differences

In some cases, library differences will be pointed out. Examples such as this:

```
#ifndef USING_PYTHON2
  return urllib.quote(str(s), '')           # python2 code
#else /* USING_PYTHON2 */
  return urllib.parse.quote(str(s), '')     # python3 code
#endif /* USING_PYTHON2 */
```

to this:

```
if USING_PYTHON2:
  return urllib.quote(str(s), '')
else:
  return urllib.parse.quote(str(s), '')
```

Sometimes this can also be done using an inline if statement, as in:

```
  return urllib.quote(str(s), '') if USING_PYTHON2 else urllib.parse.quote(str(s), '')
```

It is your decision as to which is more readable.

Octal Constants

In Python2, octal constants can be prefixed with either "0" or "0o". In Python3, "0o" is mandated.

Code can use the new form exclusively. Change this

```
#ifndef USING_PYTHON2
  os.umask(077)     # python2 code
#else /* USING_PYTHON2 */
  os.umask(0o77)     # python3 code
#endif /* USING_PYTHON2 */
```

to just the new form:

```
  os.umask(0o77)
```

has_key() vs 'in'

The has_key() method has been removed from Python3. Fortunately, the `in` keyword works in both languages. So change this

```
#ifndef USING_PYTHON2
  if openstack.has_key('custom_configuration'):     # python2 code
#else /* USING_PYTHON2 */
  if 'custom_configuration' in openstack:     # python3 code
#endif /* USING_PYTHON2 */
```

to only use the `in` keyword, as in this:

```
  if 'custom_configuration' in openstack:
```

Print Statements vs Print Functions

The format of print statements have changed into functions in Python3. Because of the import statement we added in the Preparatory Steps above, we can use the print function format everywhere. Change this:

```
#ifndef USING_PYTHON2
  print >>fp, binascii.hexlify(b).decode('utf-8')     # python2 code
#else /* USING_PYTHON2 */
  print(binascii.hexlify(b).decode('utf-8'), file=fp)     # python3 code
#endif /* USING_PYTHON2 */
```

to only use the `print()` function, as in this:

```
  print(binascii.hexlify(b).decode('utf-8'), file=fp)
```

Dictionary .keys() and .items() Return Iterators in Python3

In Python3, the dictionary `.keys()` and `.items()` methods return iterators instead of lists. There are two cases to be considered. In some cases a list must absolutely be used, such as passing to a format conversion. So the iterator can be converted to a list. This can be simplified even further by noting that converting a list to a list is a no-op, so the Python3 form of the code can be used in both languages. For example, change this:

```
#ifndef USING_PYTHON2

  debug("ctx node has the following Properties: {}".format(i.properties.keys()))     # python2 code

#else /* USING_PYTHON2 */

  debug("ctx node has the following Properties: {}".format(list(i.properties.keys())))     # python3 code

#endif /* USING_PYTHON2 */
```

to this:

```
  debug("ctx node has the following Properties: {}".format(list(i.properties.keys())))
```

When used in for loops, the two languages are compatible for the return of `.keys()` and `.items()`. So the Python2 form of the code can be used in both languages. For example, change this:

```
#ifndef USING_PYTHON2

  for k, nv in want.items():

#else /* USING_PYTHON2 */

  for k, nv in list(want.items()):

#endif /* USING_PYTHON2 */
```

to just the Python2 form:

```
  for k, nv in want.items():
```

Additional Differences

All of the code samples above are meant to be examples of types of issues that have been discovered so far. Be sure to search the file for all instances of `#ifndef` and fix each occurrence.

If you discover additional types of changes that need to be discussed on this wiki pagr, please add them.

Clean Up Preparatory Statements

If there are no `print()` function calls in the code, this line can be removed from the Preparatory Statements at the top:

~~from __future__ import print_function~~

If there are no uses of USING_PYTHON2 in your code, this line can be removed from the top:

~~USING_PYTHON2 = sys.version_info[0] < 3~~

If there are no uses of "sys." in your code, you can also remove this line from the top:

~~import sys~~

Make Sure that Python2 AND Python3 Tests Work

To change your tox tests so that you are testing against both Python2 and Python3, a small change is needed to your `tox.ini` file. In particular, change this block from

```
[tox]
envlist = py27
```

to instead read

```
[tox]
envlist = py27,py36
```

Now run "`tox test`" to execute your test modules once with Python2 and then again with Python3. (The final choice of py34 vs py36 vs py37 is yet to be determined.)

What to do When Python3 Tests Fail

If your Python3 tests fail within your code, your recourse is to fix your code. Just do it in a way that is compatible with both versions of Python.

Most of the cloudify mock libraries are compatible with Python3, but a few are not. If your Python3 tests fail because of an incompatibility in the cloudify mock libraries, until we get modified versions of the mock libraries we won't be able to run the tests with both versions of Python. You may be able to temporarily comment out some specific mock references and test the rest.