Ansible and Python 3

The `ansible-core` code runs on both Python 2 and Python 3 because we want Ansible to be able to manage a wide variety of machines. Contributors to ansible-core and to Ansible Collections should be aware of the tips in this document so that they can write code that will run on the same versions of Python as the rest of Ansible.

To ensure that your code runs on Python 3 as well as on Python 2, learn the tips and tricks and idioms described here. Most of these considerations apply to all three types of Ansible code:

1. controller-side code - code that runs on the machine where you invoke :command:`/usr/bin/ansible`

   > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst`, **line 16**); ***backlink***
   >
   > Unknown interpreted text role "command".

2. modules - the code which Ansible transmits to and invokes on the managed machine.
3. shared `module_utils` code - the common code that's used by modules to perform tasks and sometimes used by controller-side code as well

However, the three types of code do not use the same string strategy. If you're developing a module or some `module_utils` code, be sure to read the section on string strategy carefully.

Minimum version of Python 3.x and Python 2.x

On the controller we support Python 3.5 or greater and Python 2.7 or greater. Module-side, we support Python 3.5 or greater and Python 2.6 or greater.

Python 3.5 was chosen as a minimum because it is the earliest Python 3 version adopted as the default Python by a Long Term Support (LTS) Linux distribution (in this case, Ubuntu-16.04). Previous LTS Linux distributions shipped with a Python 2 version which users can rely upon instead of the Python 3 version.

For Python 2, the default is for modules to run on at least Python 2.6. This allows users with older distributions that are stuck on Python 2.6 to manage their machines. Modules are allowed to drop support for Python 2.6 when one of their dependent libraries requires a higher version of Python. This is not an invitation to add unnecessary dependent libraries in order to force your module to be usable only with a newer version of Python; instead it is an acknowledgment that some libraries (for instance, boto3 and docker-py) will only function with a newer version of Python.

> **Note**
>
> Python 2.4 Module-side Support:
>
> Support for Python 2.4 and Python 2.5 was dropped in Ansible-2.4. RHEL-5 (and its rebuilds like CentOS-5) were supported until April of 2017. Ansible-2.3 was released in April of 2017 and was the last Ansible release to support Python 2.4 on the module-side.

Developing Ansible code that supports Python 2 and Python 3

The best place to start learning about writing code that supports both Python 2 and Python 3 is Lennart Regebro's book: Porting to Python 3. The book describes several strategies for porting to Python 3. The one we're using is to support Python 2 and Python 3 from a single code base

Understanding strings in Python 2 and Python 3

Python 2 and Python 3 handle strings differently, so when you write code that supports Python 3 you must decide what string model to use. Strings can be an array of bytes (like in C) or they can be an array of text. Text is what we think of as letters, digits, numbers, other printable symbols, and a small number of unprintable "symbols" (control codes).

In Python 2, the two types for these (:class:`str <python:str>` for bytes and :func:`unicode <python:unicode>` for text) are often used interchangeably. When dealing only with ASCII characters, the strings can be combined, compared, and converted from one type to another automatically. When non-ASCII characters are introduced, Python 2 starts throwing exceptions due to not knowing what encoding the non-ASCII characters should be in.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 67); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 67); *backlink*
>
> Unknown interpreted text role "func".

Python 3 changes this behavior by making the separation between bytes (:class:`bytes <python3:bytes>`) and text (:class:`str <python3:str>`) more strict. Python 3 will throw an exception when trying to combine and compare the two types. The programmer has to explicitly convert from one type to the other to mix values from each.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 74); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 74); *backlink*
>
> Unknown interpreted text role "class".

In Python 3 it's immediately apparent to the programmer when code is mixing the byte and text types inappropriately, whereas in Python 2, code that mixes those types may work until a user causes an exception by entering non-ASCII input. Python 3 forces programmers to proactively define a strategy for working with strings in their program so that they don't mix text and byte strings unintentionally.

Ansible uses different strategies for working with strings in controller-side code, in :ref:`modules <module_string_strategy>`, and in :ref:`module_utils <module_utils_string_strategy>` code.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 85); *backlink*
>
> Unknown interpreted text role "ref".

Controller string strategy: the Unicode Sandwich

In controller-side code we use a strategy known as the Unicode Sandwich (named after Python 2's :func:`unicode <python:unicode>` text type). For Unicode Sandwich we know that at the border of our code and the outside world (for example, file and network IO, environment variables, and some library calls) we are going to receive bytes. We need to transform these bytes into text and use that throughout the internal portions of our code. When we have to send those strings back out to the outside world we first convert the text back into bytes. To visualize this, imagine a 'sandwich' consisting of a top and bottom layer of bytes, a layer of conversion between, and all text type in the center.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 93); *backlink*
>
> Unknown interpreted text role "func".

Unicode Sandwich common borders: places to convert bytes to text in controller code

This is a partial list of places where we have to convert to and from bytes when using the Unicode Sandwich string strategy. It's not exhaustive but it gives you an idea of where to watch for problems.

Reading and writing to files

In Python 2, reading from files yields bytes. In Python 3, it can yield text. To make code that's portable to both we don't make use of Python 3's ability to yield text but instead do the conversion explicitly ourselves. For example:

```
from ansible.module_utils.common.text.converters import to_text

with open('filename-with-utf8-data.txt', 'rb') as my_file:
    b_data = my_file.read()
    try:
        data = to_text(b_data, errors='surrogate_or_strict')
    except UnicodeError:
        # Handle the exception gracefully -- usually by displaying a good
        # user-centric error message that can be traced back to this piece
```

```
        # of code.
        pass
```

> **Note**
>
> Much of Ansible assumes that all encoded text is UTF-8. At some point, if there is demand for other encodings we may change that, but for now it is safe to assume that bytes are UTF-8.

Writing to files is the opposite process:

```
from ansible.module_utils.common.text.converters import to_bytes

with open('filename.txt', 'wb') as my_file:
    my_file.write(to_bytes(some_text_string))
```

Note that we don't have to catch :exc:`UnicodeError` here because we're transforming to UTF-8 and all text strings in Python can be transformed back to UTF-8.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 144); *backlink*
>
> Unknown interpreted text role "exc".

Filesystem interaction

Dealing with filenames often involves dropping back to bytes because on UNIX-like systems filenames are bytes. On Python 2, if we pass a text string to these functions, the text string will be converted to a byte string inside of the function and a traceback will occur if non-ASCII characters are present. In Python 3, a traceback will only occur if the text string can't be decoded in the current locale, but it's still good to be explicit and have code which works on both versions:

```
import os.path

from ansible.module_utils.common.text.converters import to_bytes

filename = u'/var/tmp/くらとみ.txt'
f = open(to_bytes(filename), 'wb')
mtime = os.path.getmtime(to_bytes(filename))
b_filename = os.path.expandvars(to_bytes(filename))
if os.path.exists(to_bytes(filename)):
    pass
```

When you are only manipulating a filename as a string without talking to the filesystem (or a C library which talks to the filesystem) you can often get away without converting to bytes:

```
import os.path

os.path.join(u'/var/tmp/café', u'くらとみ')
os.path.split(u'/var/tmp/café/くらとみ')
```

On the other hand, if the code needs to manipulate the filename and also talk to the filesystem, it can be more convenient to transform to bytes right away and manipulate in bytes.

> **Warning**
>
> Make sure all variables passed to a function are the same type. If you're working with something like :func:`python3:os.path.join` which takes multiple strings and uses them in combination, you need to make sure that all the types are the same (either all bytes or all text). Mixing bytes and text will cause tracebacks.
>
> > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 187); *backlink*
> >
> > Unknown interpreted text role "func".

Interacting with other programs

Interacting with other programs goes through the operating system and C libraries and operates on things that the UNIX kernel defines. These interfaces are all byte-oriented so the Python interface is byte oriented as well. On both Python 2 and Python 3, byte strings should be given to Python's subprocess library and byte strings should be expected back from it.

One of the main places in Ansible's controller code that we interact with other programs is the connection plugins' `exec_command` methods. These methods transform any text strings they receive in the command (and arguments to the command) to execute into bytes and return stdout and stderr as byte strings Higher level functions (like action plugins' `_low_level_execute_command`) transform the output into text strings.

Module string strategy: Native String

In modules we use a strategy known as Native Strings. This makes things easier on the community members who maintain so many of Ansible's modules, by not breaking backwards compatibility by mandating that all strings inside of modules are text and converting between text and bytes at the borders.

Native strings refer to the type that Python uses when you specify a bare string literal:

```
"This is a native string"
```

In Python 2, these are byte strings. In Python 3 these are text strings. Modules should be coded to expect bytes on Python 2 and text on Python 3.

Module_utils string strategy: hybrid

In `module_utils` code we use a hybrid string strategy. Although Ansible's `module_utils` code is largely like module code, some pieces of it are used by the controller as well. So it needs to be compatible with modules and with the controller's assumptions, particularly the string strategy. The module_utils code attempts to accept native strings as input to its functions and emit native strings as their output.

In `module_utils` code:

- Functions **must** accept string parameters as either text strings or byte strings.
- Functions may return either the same type of string as they were given or the native string type for the Python version they are run on.
- Functions that return strings **must** document whether they return strings of the same type as they were given or native strings.

Module-utils functions are therefore often very defensive in nature. They convert their string parameters into text (using `ansible.module_utils.common.text.converters.to_text`) at the beginning of the function, do their work, and then convert the return values into the native string type (using `ansible.module_utils.common.text.converters.to_native`) or back to the string type that their parameters received.

Tips, tricks, and idioms for Python 2/Python 3 compatibility

Use forward-compatibility boilerplate

Use the following boilerplate code at the top of all python files to make certain constructs act the same way on Python 2 and Python 3:

```
# Make coding more python3-ish
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type
```

`__metaclass__ = type` makes all classes defined in the file into new-style classes without explicitly inheriting from :class:`object <python3:object>`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 269); *backlink*
>
> Unknown interpreted text role "class".

The `__future__` imports do the following:

absolute_import: Makes imports look in :data:`sys.path <python3:sys.path>` for the modules being imported, skipping the directory in which the module doing the importing lives. If the code wants to use the directory in which the module doing the importing, there's a new dot notation to do so.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 274); *backlink*
>
> Unknown interpreted text role "data".

division: Makes division of integers always return a float. If you need to find the quotient use `x // y` instead of `x / y`.

print_function: Changes :func:`print <python3:print>` from a keyword into a function.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 280); *backlink*
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 282)
>
> Unknown directive type "seealso".
>
> ```
> .. seealso::
>     * `PEP 0328: Absolute Imports <https://www.python.org/dev/peps/pep-0328/#guido-s-decision>`_
>     * `PEP 0238: Division <https://www.python.org/dev/peps/pep-0238>`_
>     * `PEP 3105: Print function <https://www.python.org/dev/peps/pep-3105>`_
> ```

Prefix byte strings with `b_`

Since mixing text and bytes types leads to tracebacks we want to be clear about what variables hold text and what variables hold bytes. We do this by prefixing any variable holding bytes with `b_`. For instance:

```
filename = u'/var/tmp/café.txt'
b_filename = to_bytes(filename)
with open(b_filename) as f:
    data = f.read()
```

We do not prefix the text strings instead because we only operate on byte strings at the borders, so there are fewer variables that need bytes than text.

Import Ansible's bundled Python `six` library

The third-party Python six library exists to help projects create code that runs on both Python 2 and Python 3. Ansible includes a version of the library in module_utils so that other modules can use it without requiring that it is installed on the remote system. To make use of it, import it like this:

```
from ansible.module_utils import six
```

> **Note**
>
> Ansible can also use a system copy of six
>
> Ansible will use a system copy of six if the system copy is a later version than the one Ansible bundles.

Handle exceptions with `as`

In order for code to function on Python 2.6+ and Python 3, use the new exception-catching syntax which uses the `as` keyword:

```
try:
    a = 2/0
except ValueError as e:
    module.fail_json(msg="Tried to divide by zero: %s" % e)
```

Do **not** use the following syntax as it will fail on every version of Python 3:

> **System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 339)**
>
> Cannot analyze code. No Pygments lexer found for "none".
>
> ```
> .. code-block:: none
>
>     try:
>         a = 2/0
>     except ValueError, e:
>         module.fail_json(msg="Tried to divide by zero: %s" % e)
> ```

Update octal numbers

In Python 2.x, octal literals could be specified as `0755`. In Python 3, octals must be specified as `0o755`.

String formatting for controller code

Use `str.format()` for Python 2.6 compatibility

Starting in Python 2.6, strings gained a method called `format()` to put strings together. However, one commonly used feature of `format()` wasn't added until Python 2.7, so you need to remember not to use it in Ansible code:

```
# Does not work in Python 2.6!
new_string = "Dear {}, Welcome to {}".format(username, location)

# Use this instead
new_string = "Dear {0}, Welcome to {1}".format(username, location)
```

Both of the format strings above map positional arguments of the `format()` method into the string. However, the first version doesn't work in Python 2.6. Always remember to put numbers into the placeholders so the code is compatible with Python 2.6.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 375)**
>
> Unknown directive type "seealso".
>
> ```
> .. seealso::
>     Python documentation on format strings:
>
>     - `format strings in 2.6 <https://docs.python.org/2.6/library/string.html#formatstrings>`_
>     - `format strings in 3.x <https://docs.python.org/3/library/string.html#formatstrings>`_
> ```

Use percent format with byte strings

In Python 3.x, byte strings do not have a `format()` method. However, it does have support for the older, percent-formatting.

```
b_command_line = b'ansible-playbook --become-user %s -K %s' % (user, playbook_file)
```

> **Note**

Percent formatting added in Python 3.5

Percent formatting of byte strings was added back into Python 3 in 3.5. This isn't a problem for us because Python 3.5 is our minimum version. However, if you happen to be testing Ansible code with Python 3.4 or earlier, you will find that the byte string formatting here won't work. Upgrade to Python 3.5 to test.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 399)

Unknown directive type "seealso".

```
.. seealso::
    Python documentation on `percent formatting <https://docs.python.org/3/library/stdtypes.html#string-for
```

Testing modules on Python 3

Ansible modules are slightly harder to code to support Python 3 than normal code from other projects. A lot of mocking has to go into unit testing an Ansible module, so it's harder to test that your changes have fixed everything or to to make sure that later commits haven't regressed the Python 3 support. Review our :ref:`testing <developing_testing>` pages for more information.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_python_3.rst, line 407); *backlink*

Unknown interpreted text role "ref".