Experience Porting to Python 3

Larsoft Coordination Meeting
Feb. 11, 2020

H. Greenlee

Contents

Introduction

- What this talk is about.
  - This talk is based on my experience in porting python scripts in the larbatch package (including project.py).
  - Also my experience in porting a c++ extension module (fcl module) in the ubutil package.
- What this talk is not.
  - This talk is not a comprehensive list of python 2 and python 3 differences (for that see resources below).
- My goal for all of these porting exercises was to produce single-source python 2/3 agnostic packages that worked in both python 2 and python 3.
- Resources.
  - https://www.python.org/
  - http://python-future.org/

Overview of Porting Process

- There are many breaking changes between python 2 and python 3.
  - Many python 3-style constructs are already available in python 2.7. These can be immediately adopted for python 2/3 agnostic code.
    - E.g. unicode strings.
  - In other cases, reasonable workarounds and idioms are available that work in both python 2 and python 3.
    - E.g. using range instead of xrange.
  - In still other cases, python 3-style constructs can be made available in python 2 using the “future” module.
    - E.g. python 3 style print function.
  - Python 2/3 conditional code.
    - Only needed in pure python code because of library module renaming.
    - Sometimes needed in C/C++.

Automatic Conversion Tool – 2to3

- Python 2.7 and python 3 both include the automatic convertion program 2to3 (works the same in either python version).
- Invoking 2to3.
  - Simple invocation, outputs unified diff patch to standard ouitput.
    - 2to3 myscript.py
  - Output patch and apply patch in situ.
    - 2to3 -w myscript.py
  - Listing available automatic conversions.
    - 2to3 -l
  - Choosing automatic conversions.
    - 2to3 -f … myscript.py
  - Excluding automatic conversions.
    - 2to3 -x … myscript.py

2to3 Example Output

```
$ 2to3 rootstat.py
RefactoringTool: Skipping optional fixer: buffer
RefactoringTool: Skipping optional fixer: idioms
RefactoringTool: Skipping optional fixer: set_literal
RefactoringTool: Skipping optional fixer: ws_comma
RefactoringTool: Refactored rootstat.py
--- rootstat.py (original)
+++ rootstat.py (refactored)
@@ -36,7 +36,7 @@

 myargv = sys.argv
 sys.argv = myargv[0:1]
-if os.environ.has_key('TERM'):
+if 'TERM' in os.environ:
     del os.environ['TERM']
 import ROOT
 ROOT.gErrorIgnoreLevel = ROOT.kError
@@ -58,9 +58,9 @@
             doprint = 0
         if doprint:
             if len(line) > 2:
-                print line[2:],
-            else:
-                print
+                print(line[2:], end=' ')
+            else:
+                print()

 # Analyze root file.

@@ -71,7 +71,7 @@
     keys = root.GetListOfKeys()
     for key in keys:
         objname = key.GetName()
-        if not trees.has_key(objname):
+        if objname not in trees:
             obj = root.Get(objname)
             if obj and obj.InheritsFrom('TTree'):
                 trees[objname] = obj
```

How to Use 2to3

- 2to3 generates valid python 3 code starting from python 2 code.
  - Not guaranteed to be backward compatible with python 2 nor python 2/3 agnostic.
  - 2to3 doen't generate compatibility constructs based on future module.
    - In fact, it removes those unless you disable this feature using “-x future.”
- Don't blindly invoke 2to3 and accept all updates.
- Do use 2to3 to discover conversion issues in your code.
  - Exclude already-resolved issues using option -x.
  - Apply automatic conversions one at a time (use option -f), or handle updates manually.

Available 2to3 Conversions

```
$ 2to3 -l
Available transformations for the-f/--fix option:
apply
asserts
basestring
buffer
dict
except
exec
execfile
exitfunc
filter
funcattrs
future
getcwdu
has_key
idioms
import
imports
imports2
input
intern
isinstance
itertools
itertools_imports
long
map
metaclass
methodattrs
ne
next
nonzero
numliterals
operator
paren
print
raise
raw_input
reduce
renames
repr
set_literal
standarderror
sys_exc
throw
tuple_params
types
unicode
urllib
ws_comma
xrange
xreadlines
zip
```

Python 2/3 Differences I
Print Statement/Function

- The first thing you will notice when porting python 2 to python 3 is differences related to the print statement.
  - In python 3, the print statement is replaced by the built in print function (enclose arguments in parentheses).
    - Use “2to3 -w -f print” to do this automatically.
  - In python 2/3 agnostic code, add the following future import.
    - `from __future__ import print_function`
    - IMO, this is the only useful and necessary future module import.

```
$ python
Python 2.7.15 (default, Jan 11 2019, 11:17:43)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 1,2
1 2
>>> print(1,2)
(1, 2)
>>> from __future__ import print_function
>>> print(1,2)
1 2
```

Python 2/3 Differences II
Renamed Library Modules

- Several standard library modules have changed their names between python 2 and python 3. Some common examples:
  - Queue → queue
  - StringIO → io
  - Tkinter → tkinter
  - urllib → urllib.request
- I did not notice any instances where module content or api changed between python 2 and 3, but only module name.
- In python 2/3 agnostic code, put imcompatible imports in a try/except block:

```
try:
    import queue
except ImportError:
    import Queue as queue
```

Python 2/3 Differences III
Byte vs. Unicode Strings

- Byte vs. unicode string issues are the number one cause for surprising or unexpected python 2/3 incompatibilities.
  - You need to educate yourself about this issue.
- Python 2.7 and python 3 both support byte and unicode strings.
  - The difference between python 2.7 and python 3 is that the default python string type str is bytes in python 2 and unicode in python 3.
  - In python 3, it may be necessary to do explicit conversions between byte and unicode strings, which is hardly ever necessary in python 2.
- In python 2/3 agnostic code, you should consider that there are three distinct string types.
  - Default python string, type str (literal 'abc').
  - Type bytes (literal b'abc').
  - Type unicode (literal u'abc').

Python 2 and 3 String Examples

```
$ python
Python 2.7.15 (default, Jan 11 2019, 11:17:43)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> type('')
<type 'str'>
>>> type(b'')
<type 'str'>
>>> type(u'')
<type 'unicode'>
```

```
$ python
Python 3.7.2 (default, Jan 11 2019, 14:38:58)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> type('')
<class 'str'>
>>> type(b'')
<class 'bytes'>
>>> type(u'')
<class 'str'>
```

Converting Between Bytes and Unicode

- Converting between bytes and unicode works the same in python 2 and python 3.
  - Use method “decode” to convert bytes to unicode.
  - Use method “encode” to convert unicode to bytes.

```
$ python
Python 3.7.2 (default, Jan 11 2019, 14:38:58)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> type(b''.decode())
<class 'str'>
>>> type(u''.encode())
<class 'bytes'>
```

- In python 3, there are no default or implicit conversions between byte and unicode strings.
  - This implies that byte and unicode strings are always unequal.

```
>>> b'abc' == u'abc'
False
```

About Unicode Encodings

- The interpretation of any unicode string depends on its “encoding.”
- The default encoding in python 3 (and in most situations) is “utf-8.”
  - In utf-8, ASCII characters are represented by a single byte with value 0-127. Non-ASCII characters are represented using up to four bytes.
- Unicode objects do not know their own encoding. When required, the encoding must be specified externally.
  - Methods encode() and decode(), as well as several other functions and methods, accept an optional argument to specify the encoding. Default is always “utf-8.”
- I think it is safe to assume that there are few, if any, use cases for encodings other than utf-8.

String Conversion Use Cases

- Bytes to unicode or vice versa.
  - Use decode or encode.
- Convert string of unknown type to bytes.

```
if type(unknown_string) == type(b''):
    byte_string = unknown_string
elif type(unknown_string) == type(u''):
    byte_string = s.encode()
```

- Convert string of unknown type to default type str (python 2/3 agnostic).

```
if type(unknown_string) == type(''):
    default_string = unknown_string
elif type(unknown_string) == type(u''):
    default_string = s.encode()
elif type(unknown_string) == type(b''):
    default_string = s.decode()
```

When are String Conversions Necessary?

- Byte strings do not print nicely in python 3. You need to decode.

```
>>> print(b'abc')
b'abc'
>>> print(b'abc'.decode())
abc
```

- Subprocess communication (subprocess module).
  - Output obtained from subprocesses via functions subprocess.check_output or subprocess.Popen.communicate is generally returned as byte strings.
  - Also input sent to subprocesses should be sent as byte string.
- Files opened in binary mode (not an issue when files are opened in text mode).
- Web content (obtained by whatever method) is generally returned as byte string.
- C/c++ interfacing. C/c++ strings are generally byte strings.

String Advice

- In python 3 or python 2/3 agnostic code, use the default string type (type str) internally.
  - I.e., don't try to do everything using byte strings.
- Be sure to open data files in the correct mode (text or binary).
  - Text mode is the default.
- Define functions for complicated string conversions (e.g. conversions involving unknown types).
- In python 2/3 agnostic code, it is safest to accept string input of either string type.
- Use explicit string conversions at the boundary of your python universe.

Accessing Web Content

- I had some code that used module "pycurl" to access web content.
  - Pycurl is not included in standard python ups products (at Fermilab). Pycurl is a separate ups product.
  - There is currently no python 3 version of pycurl.
- I am aware that some people also like to use module "requests."
  - Also not part of standard python. Available as a separate ups product, or using "pip install."
- However, my preference is now to use module "urllib."
  - Included in standard python for python 2 and python 3.
  - For python 2/3 agnostic code, you will need a try/except block.

```
try:
    import urllib.request as urlrequest
except ImportError:
    import urllib as urlrequest
```

Database Programming

- I haven't tried database programming in python 3 (yet).
- When converting a database python program for whatever database (postgres, mysql, sqlite), make sure that you understand whether the database api returns and accepts bytes or unicode string.

Interfacing with C/C++

- String conversion issues.
- Extension modules.
- Linking against python.

Python 2/3 C/C++ String Conversions

- Converting a python string of either type to c++ string.

```
#include “Python.h”

PyObject* obj;  // Python string (input).
std::string s;  // c++ string (output).
if(PyBytes_Check(obj))
  s = std::string(PyBytes_AsString(obj));
else if(PyUnicode_Check(obj)) {
  PyObject* bytes = PyUnicode_AsUTF8String(obj);
  s = std::string(PyBytes_AsString(bytes));
}
```

- Converting c++ string to python default string.

```
#include “Python.h”

std::string s;       // c++ string (input).
PyObject* obj = 0;   // Python string (output).
#if PY_MAJOR_VERSION >= 3
  obj = PyUnicode_FromString(s.c_str());
#else
  obj = PyBytes_FromString(s.c_str());
#endif
```

Extension Modules

- Python 2 and python 3 extension modules require different initialization (need conditional compilation).

```
#if PY_MAJOR_VERSION >= 3

static struct PyModuleDef fclmodule = {
  PyModuleDef_HEAD_INIT,
  "fcl",               // Module name.
  0,                   // Module documentation.
  -1,                  // Module state size (this module has no state).
  fclmodule_methods    // Method table.
};

PyMODINIT_FUNC
PyInit_fcl(void)
{
  return PyModule_Create(&fclmodule);
}

#else

extern "C" {
  void initfcl()
  {
    Py_InitModule("fcl", fclmodule_methods);
  }
}

#endif
```

Linking C/C++ Against Python 2/3

- Python 2 and python 3 require differet link libraries.
  - Use program python-config to discover link libraries.

```
$ python-config --libs
-lpython2.7 -lpthread -ldl -lutil -lm

$ python-config --libs
-lpython3.7m -lpthread -ldl -lutil -lm
```

- You'll need some kind of conditional in CMakeLists.txt (or invoke python-config).

```
SET ( PYTHON_VERSION $ENV{PYTHON_VERSION} )
if(PYTHON_VERSION MATCHES v3)
  find_library(PYTHON NAMES python3.7m PATHS $ENV{PYTHON_LIB})
else()
  find_library(PYTHON NAMES python2.7 PATHS $ENV{PYTHON_LIB})
endif()

art_make( LIBRARY_NAME fcl
          LIB_LIBRARIES ${FHICLCPP}
                        cetlib
                        cetlib_except
                        ${PYTHON} )
```

Status of Larbatch and Project.py

- Last week's integration release included a new version of larbatch, v01_52_00 that was supposed to be python 2 and python 3 compliant.
- I did make some additional updates since then, which updates are merged to develop branch of main (github) larbatch repo. Should be included in this week's larsoft integration release.
- Going forward, the larbatch ups product is supposed to be python 2/3 agnostic. There is no “py2” or “py3” qualifier.
- Experiments may need to update their supporting python code (e.g. python module experiment_utilities.py).