Enum Lessons Learned

David C. Stauffer

June 16, 2015

Abstract

This document captures the lessons learned from using Python enums in the GHAP code.

Revision History

Revision	Date	Author(s)	Description
–	2015-04-22	DCS	Initial Release.
A	2015-05-20	DCS	Added compatibility to Python v2.7 and v3.4+.
B	2015-06-16	DCS	Converted to PDF using LaTeX.

1 Enum Details

1.1 Python Enums

First let's define an enumerator, or enum. Enums were introduced to Python in v3.4 and then back ported a long way back. They are incredibly useful, and I'm really surprised it took them that long to get around to it. The Python enum is defined entirely in the enum.py file in the lib folder and is only about 525 lines. It's also pure python without any C or other compiled libraries, so it's easy to dig into the details.

1.2 Defining Enums

```
from enum import IntEnum
class TbStatus(IntEnum):
    r"""
    Enumerator definitions for the possible Tuberculosis infection status.

    Notes
    -----
    #.  Negative values are uninfected, positive values are infected, zero
        is undefined.
    """
    null           =  0 # not set, used for preallocation
    uninfected     = -1 # never been infected
    recovered      = -2 # currently uninfected, but has been infected in the past
    latent_recent  =  1 # recently infected (<2 years)
    latent_remote  =  2 # immune stabilized infection
    active_untreat =  3 # active TB, not on treatment, or on ineffective treatment
    active_treated =  4 # active TB, on effective treatment
```

The first lesson is that if you ever really care about the numeric value of the enum, then you should use IntEnum instead of Enum. With a pure Enum, you can't do any numeric comparisons.

So for instance, if you want to find anyone who is infected, then with the IntEnum, and carefully chosen values, such that all infected states are greater than zero, then you can just do this:

```
    ix_infected = tb_status > 0
    ix_uninfected = tb_status < 0
```

You could define functions to do this for you, but the simple greater than or less than comparison will be much more efficient, and won't have to be updated if you add more states later.

```
def get_those_infected(tb_status):
    r"""
    Finds anyone who is infected with TB.
    """
    ix_infected = (tb_status == TbStatus.latent_recent) | (tb_status == TbStatus.latent_remote) | \
        (tb_status == TbStatus.active_treated) | (tb_status == TbStatus.active_untreat)
    return ix_infected
```

```
def get_those_uninfected(tb_status):
    r"""
    Finds anyone who is not infected with TB.
    """
    ix_uninfected = (tb_status == TbStatus.uninfected) | (tb_status == TbStatus.recovered)
    return ix_uninfected
```

1.3 Printing Enums

All python objects define a repr() response in the method __repr__. Some objects also define a __str__ method to be used by str() or implicitly called by print(). If the __str__ method is not defined, then __repr__ is called instead.

However, for enums, there are two levels of printing that we care about. One on the enum value, and one on the enum class. So with repr and str, there are four levels of printing that we care about.

```
    print(TbStatus2.uninfected)
    print(repr(TbStatus2.uninfected))
    print(TbStatus2)
    print(repr(TbStatus2))
```

In general, the repr method should be explicit, and the str method should be a more pretty print idea. In the Python enum class, the enum instance defines both:

```
class Enum(metaclass=EnumMeta):
    """Generic enumeration.

    Derive from this class to define new enumerations.

    """
. . .

    def __repr__(self):
        return "<%s.%s: %r>" % (
                self.__class__.__name__, self._name_, self._value_)

    def __str__(self):
        return "%s.%s" % (self.__class__.__name__, self._name_)
```

Thus printing these:

```
    print(TbStatus2.uninfected)
    print(repr(TbStatus2.uninfected))
```

Gives this:

```
TbStatus2.uninfected
<TbStatus2.uninfected: -1>
```

While this:

```
    print(TbStatus2)
    print(repr(TbStatus2))
```

Gives this:

```
<enum 'TbStatus2'>
<enum 'TbStatus2'>
```

The first str method doesn't tell you anything except the name you already used, however the first repr method does show something useful. It shows the name and the value.

The two print and repr methods on the class themselves only show you the name of the enum it's derived from. Also not very useful. So let's use the one useful example to extend the built-in class.

```
class _IntEnumPlus(int, Enum):
    r"""
    Custom IntEnum class based on _EnumMetaPlus metaclass to get more details
    from repr/str.
    """
    def __str__(self):
        return '{}.{}: {}'.format(self.__class__.__name__, self.name, self.value)
```

Now the print(TbStatus) becomes a clean version (i.e. no <>brackets) of the repr method.

```
TbStatus.uninfected: -1
```

In order to override the methods on the class, we actually have to modify the metaclass that it is derived from. So let's derive our own metaclass, and then derive our own enum class from that.

```
#%% Meta Class
class _EnumMetaPlus(EnumMeta):
    r"""
    Overrides the repr/str methods of the EnumMeta class to display all possible
    values.
    """
    def __repr__(cls):
        text = [repr(field) for field in cls]
        return '\n'.join(text)
    def __str__(cls):
        text = [str(field) for field in cls]
        return '\n'.join(text)

#%% Extened IntEnum class
@unique
class _IntEnumPlus(int, Enum, metaclass=_EnumMetaPlus):
    r"""
    Custom IntEnum class based on _EnumMetaPlus metaclass to get more details from
    repr/str.

    Also forces all values to be unique.
    """
    def __str__(self):
        return '{}.{}: {}'.format(self.__class__.__name__, self.name, self.value)
```

Note, that looking at the python code shows a ridiculously simple definition for IntEnum, so derive this class in the same way directly from Enum.

```
class IntEnum(int, Enum):
    """Enum where members are also (and must be) ints"""
```

Also, hidden in the enum options is a "unique" decorator that forces the names and values to be unique. In my use cases, I also want this, so I added that line to the class.

Putting this all together and printing the results:

```
    print(TbStatus)
    print(repr(TbStatus))
```

Gives this:

```
TbStatus.null: 0
TbStatus.uninfected: -1
TbStatus.recovered: -2
TbStatus.latent_recent: 1
TbStatus.latent_remote: 2
TbStatus.active_untreat: 3
TbStatus.active_treated: 4
```

And this:

```
<TbStatus.null: 0>
<TbStatus.uninfected: -1>
<TbStatus.recovered: -2>
<TbStatus.latent_recent: 1>
<TbStatus.latent_remote: 2>
<TbStatus.active_untreat: 3>
<TbStatus.active_treated: 4>
```

1.4 Backwards Compatibility

Subclassing the metaclass introduces a backwards compatibility issue into the code. In Python v2.7 and before, you would do this via:

```
class _IntEnumPlus(int, Enum):
    __metaclass__ =_EnumMetaPlus
```

In Python v3.4 and beyond you would do this via:

```
class _IntEnumPlus(int, Enum, metaclass=_EnumMetaPlus):
```

However, each form triggers an error in the other version of the language. If you want to simultaneously support both versions, then you need to use the "six" library. That library is made specifically for situations like this (the name six comes from Python 2 * 3). So now it becomes:

```
from six import with_metaclass
class _IntEnumPlus(with_metaclass(_EnumMetaPlus, int, Enum)):
```

2 Appendix

The full source code for these examples is thus:

```python
# -*- coding: utf-8 -*-
r"""
Enum lessons learned examples.

Notes
-----
#.  Written by David C. Stauffer in May 2015.
"""

#%% Imports
from __future__ import print_function
from __future__ import division
from enum import IntEnum, Enum, unique, EnumMeta
import numpy as np
from six import with_metaclass

#%% Meta Class
class _EnumMetaPlus(EnumMeta):
    r"""
    Overrides the repr/str methods of the EnumMeta class to display all possible
    values.
    """
    def __repr__(cls):
        text = [repr(field) for field in cls]
        return '\n'.join(text)
    def __str__(cls):
        text = [str(field) for field in cls]
        return '\n'.join(text)

#%% Extened IntEnum class
@unique
class _IntEnumPlus(with_metaclass(_EnumMetaPlus, int, Enum)):
    r"""
    Custom IntEnum class based on _EnumMetaPlus metaclass to get more details from
    repr/str.

    Also forces all values to be unique.
    """
    def __str__(self):
        return '{}.{}: {}'.format(self.__class__.__name__, self.name, self.value)

#%% TB Status
class TbStatus(_IntEnumPlus):
    r"""
    Enumerator definitions for the possible Tuberculosis infection status.

    Notes
    -----
    #.  Negative values are uninfected, positive values are infected, zero
        is undefined.
    """
    null           =  0 # not set, used for preallocation
    uninfected     = -1 # never been infected
    recovered      = -2 # currently uninfected, but have been infected in the past
    latent_recent  =  1 # recently infected (<2 years)
    latent_remote  =  2 # immune stabilized infection
    active_untreat =  3 # active TB, not on treatment, or on ineffective treatment
    active_treated =  4 # active TB, on effective treatment

class TbStatus2(IntEnum):
    r"""
    Standard Enumerator
    """
    null           =  0
    uninfected     = -1
    recovered      = -2
    latent_recent  =  1
    latent_remote  =  2
```

```
    active_untreat =  3
    active_treated =  4

#%% Functions
def get_those_infected(tb_status):
    r"""
    Finds anyone who is infected with TB.
    """
    ix_infected = (tb_status == TbStatus.latent_recent) | (tb_status == \
        TbStatus.latent_remote) | (tb_status == TbStatus.active_treated) | \
        (tb_status == TbStatus.active_untreat)
    return ix_infected

def get_those_uninfected(tb_status):
    r"""
    Finds anyone who is not infected with TB.
    """
    ix_uninfected = (tb_status == TbStatus.uninfected) | \
        (tb_status == TbStatus.recovered)
    return ix_uninfected

#%% Example usage
if __name__ == '__main__':
    num = 100
    tb_status = np.empty(num, dtype=int)
    tb_status.fill(TbStatus.null)
    ix = np.random.rand(num)
    tb_status[ix >= 0.5] = TbStatus.active_treated
    tb_status[ix <  0.5] = TbStatus.uninfected


    ix_infected1 = tb_status > 0
    ix_infected2 = get_those_infected(tb_status)

    ix_uninfected1 = tb_status < 0
    ix_uninfected2 = get_those_uninfected(tb_status)

    np.testing.assert_equal(ix_infected1, ix_infected2)
    np.testing.assert_equal(ix_uninfected1, ix_uninfected2)

    # normal Enums
    print('Normal')
    print(TbStatus2.uninfected)
    print(repr(TbStatus2.uninfected))
    print(TbStatus2)
    print(repr(TbStatus2))

    # extended Enums
    print('Extended')
    print(TbStatus.uninfected)
    print(repr(TbStatus.uninfected))
    print(TbStatus)
    print(repr(TbStatus))
```