All About That Constant

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

I'm not sure I've ever seen "gravitas" used as a kind of software metric. However, if there were such a thing, I think it could probably be measured by the number of predefined constants required to carry out any sort of task. For example, directly programming with sockets and setting socket options has a high degree of gravitas. Simply specifying a port number to a Web framework—not so much. Other examples might include programming OpenGL vs. turtle graphics. Or maybe just about anything involving OpenSSL. Extra bonus points are earned if such constants can get together in an unholy bitmask such as `O_RDWR | O_CREAT`. Yes, constants. Gravitas.

Constants, or shall I say "constants," have always been relatively easy to define in Python. Simply create some variables:

```
AF_UNIX = 1
AF_INET = 2
AF_IPX = 23
AF_INET6 = 30

SOCK_STREAM = 1
SOCK_DGRAM = 2
SOCK_RAW = 3
```

Then, pass these values along whenever you need to use them:

```
sock = socket(AF_INET, SOCK_STREAM)
```

Yes, it's pretty simple stuff. Of course, all of those constants are really just simple variables. And they're not constants either. Go ahead and change them if you dare:

```
AF_INET = 30
```

Alas, it's probably foolhardy to expect any modern high-level language to support the full power of C preprocessor macros (e.g., #define AF_INET 2). So, people who decide to change constants probably get what they deserve. I digress.

Problems with Constants

Gravitas aside, constants have always presented a number of weird problems for Python programmers. For example, suppose you're using Python 2.7 and you're trying to perform debugging and diagnostics. In your code, the constants are merely presented as their corresponding value. For example, consider this code:

```
from socket import socket

def create_socket(address_family, socket_type):
    log.info('Creating socket: family=%s, type=%s',
address_family, socket_type)
    return socket(address_family, socket_type)
```

If you call the function using `create_socket(AF_INET, SOCK_STREAM)`, you'll get a log message that looks like this:

```
INFO:Creating socket: family=2, type=1
```

As you can see, you lose the symbolic names such as AF_INET, putting the burden on users to perform some kind of reverse lookup if they want to find out more information. Even doing that is a bit annoying if you don't know what you're doing. For example, do you simply go through the socket module constants one-by-one in the interactive interpreter?

```
>>>AF_UNIX
1
>>>AF_INET
2
>>>SOCK_STREAM
1
>>>
```

Or, if you're really stuck, do you pull out some kind of advanced magic to see all of the possible values?

```
>>>import socket
>>> sorted((getattr(socket, name), name) for name in
dir(socket)
...        if name.startswith('AF_'))
...
[(0, 'AF_UNSPEC'), (1, 'AF_UNIX'), (2, 'AF_INET'), (11, 'AF_SNA'),
(12, 'AF_DECnet'), (16, 'AF_APPLETALK'), (17, 'AF_ROUTE'),
(23, 'AF_IPX'),
(30, 'AF_INET6')]
>>>
```

Suppose you wanted to add some kind of enforcement of constant values in your code: for example, making sure the user only provided valid values for the arguments. Maybe you would write something like this:

```
from socket import (socket,
                    AF_UNIX, AF_INET, AF_INET6,
                    SOCK_STREAM, SOCK_DGRAM)

_address_families = { AF_UNIX, AF_INET, AF_INET6 }
_socket_types = { SOCK_STREAM, SOCK_DGRAM }

def create_socket(address_family, socket_type):
    log.info('Creating socket: family=%s, type=%s',
address_family, socket_type)
    if address_family not in _address_families:
        raise ValueError('Bad address family %s' % address_family)
    if socket_type not in _socket_types:
        raise ValueError('Bad type %s' % socket_type)
    return socket(address_family, socket_type)
```

Such a solution is kind of verbose and annoying. Moreover, it only "works" until a user comes along and writes the arguments in the wrong order such as `create_socket(SOCK_STREAM, AF_INET)`. Or did they write `create_socket(AF_UNIX, SOCK_DGRAM)`? There's really no way to know. The mind boggles.

Constants in Python 3

Starting in Python 3.4, an interesting thing happened to constants. Fire up a Python 3.4 interpreter and take a look at the socket module:

```
>>> # This must be done in Python 3.4
>>>import socket
>>>socket.AF_INET
<AddressFamily.AF_INET: 2>
>>>socket.SOCK_STREAM
<SocketType.SOCK_STREAM: 1>
>>>
```

Notice how the constants now identify themselves by a symbolic name and value. This is very different. Moreover, this change affects everything else. For instance, if you print a constant, you just get the name:

```
>>>print(socket.AF_INET)
AddressFamily.AF_INET
>>>
```

This means that in other code, such as the example involving logging, you'll now get a log message that looks like this:

```
INFO:Creating socket: family=AddressFamily.AF_INET,
type=SocketType.SOCK_STREAM
```

In fact, you can even do a kind of type checking. Consider this slightly modified version of code:

```
from socket import socket, AddressFamily, SocketType

def create_socket(address_family, socket_type):
    log.info('Creating socket: family=%s, type=%s',
address_family, socket_type)
    if not isinstance(address_family, AddressFamily):
        raise TypeError('Bad address family %s' % address_family)
    if not isinstance(socket_type, SocketType):
        raise TypeError('Bad type %s' % socket_type)
    return socket(address_family, socket_type)
```

In this example, `AddressFamily` and `SocketType` represent all of the valid values for the `address_family` and `socket_type` arguments, respectively. However, this checking is more than just values. It will catch errors such as swapped arguments like this:

```
>>> # Good
>>> s = create_socket(AF_INET, SOCK_STREAM)
>>>
>>> # Bad
>>>s = create_socket(SOCK_STREAM, AF_INET)
Traceback (most recent call last):
...
TypeError: Bad address family SocketType.SOCK_STREAM

>>>
```

Keep in mind, the value of `SOCK_STREAM` is perfectly valid as an address family (it's the same as `AF_UNIX`). Yet, the code caught the error. If you're like me, you'll find all of this to be very interesting.

Enter Enums

Starting in Python 3.4, you can now start defining constants in the form of an "enumeration" class. There are two different flavors, a standard `Enum` and an `IntEnum`. Here are some examples of enum definitions:

```
from enum import Enum
class Color(Enum):
    red = 1
    blue = 2
    green = 3

from enum import IntEnum
class AddressFamily(IntEnum):
    AF_UNIX = 1
    AF_INET = 2
    AF_IPX = 23
    AF_INET6 = 30
```

The first enumeration, `Color`, simply defines a collection of symbolic constants. To refer to them in your code, you just use the class name as a prefix like this:

```
>>>Color.red
<Color.red: 1>
>>>Color.blue
<Color.blue: 2>
>>>
```

Normally, you will just use these names in your code. However, should you need to know the name and value, you can obtain them as attributes as follows:

```
>>>Color.blue.name
'blue'
>>>Color.blue.value
2
>>>AddressFamily.AF_INET.value
2
>>>
```

Such attributes can be useful in situations where you need to convert an enum into a different format or into a value that you might use in an external representation (e.g., JSON). To go the other way, you can use the class name to convert a value back into an enum:

```
>>>Color(2)
<Color.blue: 2>
>>>Color(4)
Traceback (most recent call last):
...
ValueError: 4 is not a valid Color
>>>AddressFamily(2)
<AddressFamily.AF_INET: 2>
>>>
```

As you can see, such conversions are already aware of the valid enum values. If you try to convert a bad value, you'll get an error.

If you want to know all of the possible values of an enumeration, simply turn the class into a list or iterate over it. For example:

```
>>>list(Color)
[<Color.red: 1>, <Color.blue: 2>, <Color.green: 3>]
>>>for val in AddressFamily:
...     print(val)
...
AddressFamily.AF_UNIX
AddressFamily.AF_INET
AddressFamily.AF_IPX
AddressFamily.AF_INET6
>>>
```

In this example, two different kinds of enums were defined. The difference between `Enum` and `IntEnum` concerns their interaction with the rest of the type system and compatibility with the integers.

`Enum` types implement a strict form of type checking that do not allow any kind of mixing with other types or other enums. For example:

```
>>>Color.blue
<Color.blue: 2>
>>>Color.blue == 2    # Notice failed equality
False
>>>Color.blue == AddressFamily.AF_INET
False
>>>Color.blue + 4
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Color' and 'int'
>>>
```

In fact, the values associated with an Enum type are arbitrary. For example, it would be perfectly fine to define this:

```
class Color(Enum):
    red = 'R'
    blue = 'B'
    green = 'G'
```

Keep in mind that the intended use of the enum would be through the symbolic names such as Color.red, not the value. As such, nothing is implied or guaranteed about the capabilities of the enum itself. The value really only becomes useful in code that needs to convert the enum to/from a different type for instance.

The IntEnum class, on the other hand, makes an enum compatible with integers. This is especially useful if you're defining constants that need to interface with external libraries or legacy code. Or if the constants need to be used in mathematical operations. For example:

```
>>>AddressFamily.AF_INET == 2
True
>>>AddressFamily.AF_INET + 10
12
>>>
```

IntEnum types are also useful if constants are defined in order to perform other operations such as the formation of a bitmask. For example:

```
>>>class Modes(IntEnum):
...     READ = 1
...     WRITE = 2
...     DELETE = 4
...
>>> a = Modes.READ | Modes.WRITE
>>>a
3
>>>
```

Making Enums

Perhaps the most obvious way to define an enum is through a class definition as shown in the example. However, this offers no help to existing code where a large number of constants might already exist. Fortunately, there is an alternate interface involving dictionaries. For example, suppose you have some constants already populating a dict like this:

```
colors = {
    'red' : 1,
    'blue' : 2,
    'green' : 3
}
```

To create an enum, simply call Enum() or IntEnum() as a function and pass the dictionary like this:

```
from enum import Enum
Color = Enum('Color', colors)
```

If you are clever, you can use this to create enumerations from existing sets of constants. For example, suppose you wanted to make an enum from all of the flags passed to the os.open() function. You could simply gather them up using a dictionary comprehension and pass them to IntEnum() like this:

```
>>>import os
>>>flagvals = { name:val for name, val in vars(os).items()
...                 if name.startswith('O_') }
>>>flagvals
{'O_SYNC': 128, 'O_SHLOCK': 16, 'O_TRUNC': 1024, 'O_CREAT': 512,
'O_EXCL': 2048, 'O_RDWR': 2, 'O_DSYNC': 4194304, 'O_NONBLOCK': 4,
'O_ACCMODE': 3, 'O_WRONLY': 1, 'O_ASYNC': 64, 'O_RDONLY': 0,
'O_APPEND': 8, 'O_NOFOLLOW': 256, 'O_DIRECTORY': 1048576,
'O_NOCTTY': 131072, 'O_NDELAY': 4, 'O_EXLOCK': 32}
>>>Flags = IntEnum('Flags', flagvals)
>>>Flags.O_TRUNC
<Flags.O_TRUNC: 1024>
>>>Flags.O_CREAT
<Flags.O_CREAT: 512>
>>>Flags.O_RDONLY
<Flags.O_RDONLY: 0>
>>>
```

If you were feeling particularly adventurous, you could even patch the original os module to use the newly created enums:

```
>>>vars(os).update({f.name:f for f in Flags})
>>>os.O_CREAT
<Flags.O_CREAT: 512>
>>>os.O_RDWR
<Flags.O_RDWR: 2>
>>>
```

Since IntEnum classes are compatible with integers, everything should continue to work the same as before except for symbolic names appearing in the event that a flag value is ever printed or logged.

The Normal Rules Don't Apply

Having introduced enums, most Python programmers will find them to behave in all sorts of ways that are quite different from normal class definitions. For example, duplicate entries result in an error:

```
class Color(Enum):
     red = 1
     blue = 2
     red = 3        # An error. Duplicate.
```

Enum classes always keep their entries in the same order as listed:

```
class Color(Enum):
    red = 10
    blue = 9
    green = 8

cols = list(Color)   #  [ Color.red, Color.blue, Color.green]
```

You can't inherit from an enum:

```
class MyColor(Color):
    purple = 4            # Error. Can't extend Color
```

And the members of an enum can't be redefined:

```
Color.red = 4             # Error. Can't reassign members
```

The members of an enum are also instances of the class itself:

```
>>>isinstance(Color.blue, Color)
True
>>>
```

All of this unusual behavior is the result of enums being defined through advanced features of Python metaclasses. It's not possible (or really necessary) to dive into the details here, but if you've ever wondered about the power of Python metaprogramming, enums are a good example of what's possible.

Final Words

As a new Python feature, enums are not something you're likely to encounter in much code. However, they are starting to be used in various places in the standard library and will likely have increased usage in future Python versions. In my own application code, I often find myself defining various sorts of constants to indicate modes, flags, and similar kinds of functionality. With the addition of enums, I'm now starting to think that they might be a useful way to provide improved debugging, type safety, and other similar features. Although enums first appeared in Python 3.4, the `flufl.enum` package can be used to add them to earlier versions of Python including Python 2.7.

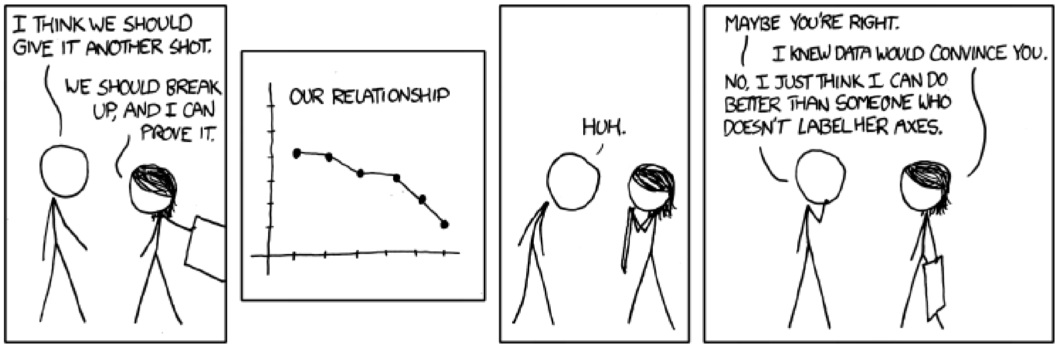
Resources

https://docs.python.org/3/library/enum.html (official documentation for enums).

http://legacy.python.org/dev/peps/pep-0435/ (adding an `Enum` type to the Python Standard Library).

https://pypi.python.org/pypi/flufl.enum (an enum implementation compatible with Python 2.7).

XKCD


I THINK WE SHOULD GIVE IT ANOTHER SHOT.
WE SHOULD BREAK UP, AND I CAN PROVE IT.
OUR RELATIONSHIP
HUH.
MAYBE YOU'RE RIGHT.
I KNEW DATA WOULD CONVINCE YOU.
NO, I JUST THINK I CAN DO BETTER THAN SOMEONE WHO DOESN'T LABEL HER AXES.

xkcd.com