enum --- support for enumerations

An enumeration is a set of symbolic names (members) bound to unique, constant values. Within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated over.

This module defines two enumeration classes that can be used to define unique sets of names and values: `Enum` and `IntEnum`. It also defines one decorator, `unique`, that ensures only unique member names are present in an enumeration.

Creating an Enum

Enumerations are created using the `class` syntax, which makes them easy to read and write. An alternative creation method is described in Functional API. To define an enumeration, subclass `Enum` as follows:

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

Note: Nomenclature

- The class `Color` is an *enumeration* (or *enum*)
- The attributes `Color.red`, `Color.green`, etc., are *enumeration members* (or *enum members*).
- The enum members have *names* and *values* (the name of `Color.red` is `red`, the value of `Color.blue` is 3, etc.)

Note:

> Even though we use the `class` syntax to create Enums, Enums are not normal Python classes. See How are Enums different? for more details.

Enumeration members have human readable string representations:

```
>>> print(Color.red)
Color.red
```

...while their `repr` has more information:

```
>>> print(repr(Color.red))
<Color.red: 1>
```

The *type* of an enumeration member is the enumeration it belongs to:

```
>>> type(Color.red)
<enum 'Color'>
>>> isinstance(Color.green, Color)
True
>>>
```

Enum members also have a property that contains just their item name:

```
>>> print(Color.red.name)
red
```

Enumerations support iteration. In Python 3.x definition order is used; in Python 2.x the definition order is not available, but class attribute `__order__` is supported; otherwise, value order is used:

```
>>> class Shake(Enum):
...   __order__ = 'vanilla chocolate cookies mint'  # only needed in 2.x
...   vanilla = 7
...   chocolate = 4
...   cookies = 9
...   mint = 3
...
>>> for shake in Shake:
...   print(shake)
...
Shake.vanilla
Shake.chocolate
Shake.cookies
Shake.mint
```

The `__order__` attribute is always removed, and in 3.x it is also ignored (order is definition order); however, in the stdlib version it will be ignored but not removed.

Enumeration members are hashable, so they can be used in dictionaries and sets:

```
>>> apples = {}
>>> apples[Color.red] = 'red delicious'
>>> apples[Color.green] = 'granny smith'
>>> apples == {Color.red: 'red delicious', Color.green: 'granny smith'}
True
```

Programmatic access to enumeration members and their attributes

Sometimes it's useful to access members in enumerations programmatically (i.e. situations where `Color.red` won't do because the exact color is not known at program-writing time). `Enum` allows such access:

```
>>> Color(1)
<Color.red: 1>
>>> Color(3)
<Color.blue: 3>
```

If you want to access enum members by *name*, use item access:

```
>>> Color['red']
<Color.red: 1>
>>> Color['green']
<Color.green: 2>
```

If have an enum member and need its `name` or `value`:

```
>>> member = Color.red
>>> member.name
'red'
>>> member.value
1
```

Duplicating enum members and values

Having two enum members (or any other attribute) with the same name is invalid; in Python 3.x this would raise an error, but in Python 2.x the second member simply overwrites the first:

```
>>> # python 2.x
>>> class Shape(Enum):
...   square = 2
...   square = 3
...
>>> Shape.square
<Shape.square: 3>

>>> # python 3.x
>>> class Shape(Enum):
...   square = 2
...   square = 3
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'square'
```

However, two enum members are allowed to have the same value. Given two members A and B with the same value (and A defined first), B is an alias to A. By-value lookup of the value of A and B will return A. By-name lookup of B will also return A:

```
>>> class Shape(Enum):
...   __order__ = 'square diamond circle alias_for_square'  # only needed in 2.x
...   square = 2
...   diamond = 1
...   circle = 3
...   alias_for_square = 2
...
>>> Shape.square
<Shape.square: 2>
>>> Shape.alias_for_square
<Shape.square: 2>
>>> Shape(2)
<Shape.square: 2>
```

Allowing aliases is not always desirable. `unique` can be used to ensure that none exist in a particular enumeration:

```
>>> from enum import unique
>>> @unique
... class Mistake(Enum):
...   __order__ = 'one two three four'  # only needed in 2.x
...   one = 1
...   two = 2
...   three = 3
```

```
...   four = 3
Traceback (most recent call last):
...
ValueError: duplicate names found in <enum 'Mistake'>: four -> three
```

Iterating over the members of an enum does not provide the aliases:

```
>>> list(Shape)
[<Shape.square: 2>, <Shape.diamond: 1>, <Shape.circle: 3>]
```

The special attribute `__members__` is a dictionary mapping names to members. It includes all names defined in the enumeration, including the aliases:

```
>>> for name, member in sorted(Shape.__members__.items()):
...   name, member
...
('alias_for_square', <Shape.square: 2>)
('circle', <Shape.circle: 3>)
('diamond', <Shape.diamond: 1>)
('square', <Shape.square: 2>)
```

The `__members__` attribute can be used for detailed programmatic access to the enumeration members. For example, finding all the aliases:

```
>>> [name for name, member in Shape.__members__.items() if member.name != name]
['alias_for_square']
```

Comparisons

Enumeration members are compared by identity:

```
>>> Color.red is Color.red
True
>>> Color.red is Color.blue
False
>>> Color.red is not Color.blue
True
```

Ordered comparisons between enumeration values are *not* supported. Enum members are not integers (but see IntEnum below):

```
>>> Color.red < Color.blue
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Color() < Color()
```

Warning

In Python 2 *everything* is ordered, even though the ordering may not make sense. If you want your enumerations to have a sensible ordering check out the OrderedEnum recipe below.

Equality comparisons are defined though:

```
>>> Color.blue == Color.red
False
>>> Color.blue != Color.red
True
>>> Color.blue == Color.blue
True
```

Comparisons against non-enumeration values will always compare not equal (again, `IntEnum` was explicitly designed to behave differently, see below):

```
>>> Color.blue == 2
False
```

Allowed members and attributes of enumerations

The examples above use integers for enumeration values. Using integers is short and handy (and provided by default by the Functional API), but not strictly enforced. In the vast majority of use-cases, one doesn't care what the actual value of an enumeration is. But if the value *is* important, enumerations can have arbitrary values.

Enumerations are Python classes, and can have methods and special methods as usual. If we have this enumeration:

```
>>> class Mood(Enum):
...   funky = 1
...   happy = 3
...
...   def describe(self):
...     # self is the member here
...     return self.name, self.value
...
...   def __str__(self):
...     return 'my custom str! {0}'.format(self.value)
...
...   @classmethod
...   def favorite_mood(cls):
...     # cls here is the enumeration
...     return cls.happy
```

Then:

```
>>> Mood.favorite_mood()
<Mood.happy: 3>
>>> Mood.happy.describe()
('happy', 3)
>>> str(Mood.funky)
'my custom str! 1'
```

The rules for what is allowed are as follows: _sunder_ names (starting and ending with a single underscore) are reserved by enum and cannot be used; all other attributes defined within an enumeration will become members of this enumeration, with the exception of *__dunder__* names and descriptors (methods are also descriptors).

Note:

If your enumeration defines `__new__` and/or `__init__` then whatever value(s) were given to the enum member will be passed into those methods. See Planet for an example.

Restricted subclassing of enumerations

Subclassing an enumeration is allowed only if the enumeration does not define any members. So this is forbidden:

```
>>> class MoreColor(Color):
...   pink = 17
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations
```

But this is allowed:

```
>>> class Foo(Enum):
...   def some_behavior(self):
...     pass
...
>>> class Bar(Foo):
...   happy = 1
...   sad = 2
...
```

Allowing subclassing of enums that define members would lead to a violation of some important invariants of types and instances. On the other hand, it makes sense to allow sharing some common behavior between a group of enumerations. (See OrderedEnum for an example.)

Pickling

Enumerations can be pickled and unpickled:

```
>>> from enum.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.tomato is loads(dumps(Fruit.tomato, 2))
True
```

The usual restrictions for pickling apply: picklable enums must be defined in the top level of a module, since unpickling requires them to be importable from that module.

> ***Warning***
>
> In order to support the singleton nature of enumeration members, pickle protocol version 2 or higher must be used. The default in Python 2.x is 0.

Functional API

The `Enum` class is callable, providing the following functional API:

```
>>> Animal = Enum('Animal', 'ant bee cat dog')
>>> Animal
<enum 'Animal'>
>>> Animal.ant
<Animal.ant: 1>
>>> Animal.ant.value
1
>>> list(Animal)
[<Animal.ant: 1>, <Animal.bee: 2>, <Animal.cat: 3>, <Animal.dog: 4>]
```

The semantics of this API resemble `namedtuple`. The first argument of the call to `Enum` is the name of the enumeration.

The second argument is the *source* of enumeration member names. It can be a whitespace-separated string of names, a sequence of names, a sequence of 2-tuples with key/value pairs, or a mapping (e.g. dictionary) of names to values. The last two options enable assigning arbitrary values to enumerations; the others auto-assign increasing integers starting with 1. A new class derived from `Enum` is returned. In other words, the above assignment to `Animal` is equivalent to:

```
>>> class Animals(Enum):
...   ant = 1
...   bee = 2
...   cat = 3
...   dog = 4
```

Pickling enums created with the functional API can be tricky as frame stack implementation details are used to try and figure out which module the enumeration is being created in (e.g. it will fail if you use a utility function in separate module, and also may not work on IronPython or Jython). The solution is to specify the module name explicitly as follows:

```
>>> Animals = Enum('Animals', 'ant bee cat dog', module=__name__)
```

Derived Enumerations

IntEnum

A variation of `Enum` is provided which is also a subclass of `int`. Members of an `IntEnum` can be compared to integers; by extension, integer enumerations of different types can also be compared to each other:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...   circle = 1
...   square = 2
...
>>> class Request(IntEnum):
...   post = 1
...   get = 2
...
>>> Shape == 1
False
>>> Shape.circle == 1
True
```

```
>>> Shape.circle == Request.post
True
```

However, they still can't be compared to standard `Enum` enumerations:

```
>>> class Shape(IntEnum):
...   circle = 1
...   square = 2
...
>>> class Color(Enum):
...   red = 1
...   green = 2
...
>>> Shape.circle == Color.red
False
```

`IntEnum` values behave like integers in other ways you'd expect:

```
>>> int(Shape.circle)
1
>>> ['a', 'b', 'c'][Shape.circle]
'b'
>>> [i for i in range(Shape.square)]
[0, 1]
```

For the vast majority of code, `Enum` is strongly recommended, since `IntEnum` breaks some semantic promises of an enumeration (by being comparable to integers, and thus by transitivity to other unrelated enumerations). It should be used only in special cases where there's no other choice; for example, when integer constants are replaced with enumerations and backwards compatibility is required with code that still expects integers.

Others

While `IntEnum` is part of the `enum` module, it would be very simple to implement independently:

```
class IntEnum(int, Enum):
    pass
```

This demonstrates how similar derived enumerations can be defined; for example a `StrEnum` that mixes in `str` instead of `int`.

Some rules:

1. When subclassing `Enum`, mix-in types must appear before `Enum` itself in the sequence of bases, as in the `IntEnum` example above.
2. While `Enum` can have members of any type, once you mix in an additional type, all the members must have values of that type, e.g. `int` above. This restriction does not apply to mix-ins which only add methods and don't specify another data type such as `int` or `str`.
3. When another data type is mixed in, the `value` attribute is *not the same* as the enum member itself, although it is equivalant and will compare equal.
4. %-style formatting: `%s` and `%r` call `Enum`'s `__str__` and `__repr__` respectively; other codes (such as `%i` or `%h` for IntEnum) treat the enum member as its mixed-in type.

   Note: Prior to Python 3.4 there is a bug in `str`'s %-formatting: `int` subclasses are printed as strings and not numbers when the `%d`, `%i`, or `%u` codes are used.

5. `str.__format__` (or `format`) will use the mixed-in type's `__format__`. If the `Enum`'s `str` or `repr` is desired use the `!s` or `!r str` format codes.

Decorators

unique

A `class` decorator specifically for enumerations. It searches an enumeration's `__members__` gathering any aliases it finds; if any are found `ValueError` is raised with the details:

```
>>> @unique
... class NoDupes(Enum):
...     first = 'one'
...     second = 'two'
...     third = 'two'
Traceback (most recent call last):
...
ValueError: duplicate names found in <enum 'NoDupes'>: third -> second
```

Interesting examples

While `Enum` and `IntEnum` are expected to cover the majority of use-cases, they cannot cover them all. Here are recipes for some different types of enumerations that can be used directly, or as examples for creating one's own.

AutoNumber

Avoids having to specify the value for each enumeration member:

```
>>> class AutoNumber(Enum):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     __order__ = "red green blue"  # only needed in 2.x
...     red = ()
...     green = ()
...     blue = ()
...
>>> Color.green.value == 2
True
```

Note:

> The *__new__* method, if defined, is used during creation of the Enum members; it is then replaced by Enum's *__new__* which is used after class creation for lookup of existing members. Due to the way Enums are supposed to behave, there is no way to customize Enum's *__new__*.

UniqueEnum

Raises an error if a duplicate member name is found instead of creating an alias:

```
>>> class UniqueEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                     "aliases not allowed in UniqueEnum:  %r --> %r"
...                     % (a, e))
...
>>> class Color(UniqueEnum):
...     red = 1
...     green = 2
...     blue = 3
...     grene = 2
Traceback (most recent call last):
...
ValueError: aliases not allowed in UniqueEnum:  'grene' --> 'green'
```

OrderedEnum

An ordered enumeration that is not based on `IntEnum` and so maintains the normal `Enum` invariants (such as not being comparable to other enumerations):

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self._value_ >= other._value_
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self._value_ > other._value_
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self._value_ <= other._value_
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self._value_ < other._value_
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     __ordered__ = 'A B C D F'
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True
```

Planet

If `__new__` or `__init__` is defined the value of the enum member will be passed to those methods:

```
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
...         self.mass = mass       # in kilograms
...         self.radius = radius   # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant  (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

How are Enums different?

Enums have a custom metaclass that affects many aspects of both derived Enum classes and their instances (members).

Enum Classes

The `EnumMeta` metaclass is responsible for providing the `__contains__`, `__dir__`, `__iter__` and other methods that allow one to do things with an `Enum` class that fail on a typical class, such as `list(Color)` or `some_var in Color`. `EnumMeta` is responsible for ensuring that various other methods on the final `Enum` class are correct (such as `__new__`, `__getnewargs__`, `__str__` and `__repr__`)

Enum Members (aka instances)

The most interesting thing about Enum members is that they are singletons. `EnumMeta` creates them all while it is creating the `Enum` class itself, and then puts a custom `__new__` in place to ensure that no new ones are ever instantiated by returning only the existing member instances.

Finer Points

Enum members are instances of an Enum class, and even though they are accessible as `EnumClass.member`, they are not accessible directly from the member:

```
>>> Color.red
<Color.red: 1>
>>> Color.red.blue
Traceback (most recent call last):
```

```
...
AttributeError: 'Color' object has no attribute 'blue'
```

Likewise, `__members__` is only available on the class.

In Python 3.x `__members__` is always an `OrderedDict`, with the order being the definition order. In Python 2.7 `__members__` is an `OrderedDict` if `__order__` was specified, and a plain `dict` otherwise. In all other Python 2.x versions `__members__` is a plain `dict` even if `__order__` was specified as the `OrderedDict` type didn't exist yet.

If you give your `Enum` subclass extra methods, like the Planet class above, those methods will show up in a *dir* of the member, but not of the class:

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS',
'VENUS', '__class__', '__doc__', '__members__', '__module__']
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

A `__new__` method will only be used for the creation of the `Enum` members -- after that it is replaced. This means if you wish to change how `Enum` members are looked up you either have to write a helper function or a `classmethod`.