`aenum` --- support for advanced enumerations, namedtuples, and constants

---

An `Enum` 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.

A `NamedTuple` is a class-based, fixed-length tuple with a name for each possible position accessible using attribute-access notation.

A `NamedConstant` is a class whose members cannot be rebound; it lacks all other `Enum` capabilities, however; consequently, it can have duplicate values. There is also a `module` function that can insert the `NamedConstant` class into `sys.modules` where it will appear to be a module whose top-level names cannot be rebound.

> ***Note***
>
> `constant` **refers to names not being rebound; mutable objects can be** mutated.

Module Contents

This module defines five enumeration classes that can be used to define unique sets of names and values, one `Enum` class decorator, one `NamedTuple` class, one `NamedConstant` class, and several helpers.

`NamedConstant`

NamedConstant class for creating groups of constants. These names cannot be rebound to other values.

`Enum`

Base class for creating enumerated constants. See section Enum Functional API for an alternate construction syntax.

`AddValue`

Flag specifying that `_generate_next_value_` should always be called to provide the initial value for an enum member.

`MultiValue`

Flag specifying that each item of tuple value is a separate value for that member; the first tuple item is the canonical one.

`NoAlias`

Flag specifying that duplicate valued members are distinct and not aliases; by-value lookups are disabled.

`Unique`

Flag specifying that duplicate valued members are not allowed.

> ***Note***
>
> The flags are inherited by the enumeration's subclasses. To use them in Python 2 assign to `_settings_` in the class body.

`IntEnum`

Base class for creating enumerated constants that are also subclasses of `int`.

`AutoNumberEnum`

Derived class that automatically assigns an `int` value to each member.

`OrderedEnum`

Derived class that adds <, <=, >=, and > methods to an `Enum`.

`UniqueEnum`

Derived class that ensures only one name is bound to any one value.

`unique`

Enum class decorator that ensures only one name is bound to any one value.

> ***Note***
>
> the `UniqueEnum` class, the `unique` decorator, and the Unique flag all do the same thing; you do not need to use more than one of them at the same time.

`NamedTuple`

Base class for creating NamedTuples, either by subclassing or via it's functional API.

`constant`

Descriptor to add constant values to an `Enum`, or advanced constants to `NamedConstant`.

`convert`

Helper to transform target global variables into an `Enum`.

`enum`

Helper for specifying keyword arguments when creating `Enum` members.

`export`

Helper for inserting `Enum` members and `NamedConstant` constants into a namespace (usually `globals()`.

`extend_enum`

Helper for adding new `Enum` members, both stdlib and aenum.

`module`

Function to take a `NamedConstant` or `Enum` class and insert it into `sys.modules` with the affect of a module whose top-level constant and member names cannot be rebound.

`skip`

Descriptor to add a normal (non-`Enum` member) attribute to an `Enum` or `NamedConstant`.

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 Enum Functional API. To define an enumeration, subclass `Enum` as follows:

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

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)
<aenum 'Color'>
>>> isinstance(Color.green, Color)
True
```

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 if posible, otherwise alphabetical name 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, but in 3.x it is also used to verify that definition order is the same (useful for py2&3 code bases); however, in the stdlib version it will be ignored and not removed.

> ***Note***
>
> To maintain compatibility with Python 3.4 and 3.5, use __order__ instead (double leading and trailing underscores).

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
```

In Python 3 the class syntax has a few extra advancements:

```
--> class Color(
...         Enum,
...         settings=(AddValue, MultiValue, NoAlias, Unique),
...         init='field_name1 field_name2 ...',
...         start=7,
...         )
...
```

`start` is used to specify the starting value for the first member:

```
--> class Count(Enum, start=11):
...     eleven
...     twelve
...
--> Count.twelve.value == 12
True
```

`init` specifies the attribute names to store creation values to:

```
--> class Planet(Enum, init='mass radius'):
...     MERCURY = (3.303e+23, 2.4397e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...
```

```
--> Planet.EARTH.value
(5.976e+24, 6378140.0)
--> Planet.EARTH.radius
2.4397e6
```

The various settings enable special behavior:

- `AddValue` calls a user supplied `_generate_next_value_` to provide the initial value
- `MultiValue` allows multiple values per member instead of the usual 1
- `NoAlias` allows different members to have the same value
- `Unique` disallows different members to have the same value

> ***Note***
>
> To use these features in Python 2 use the _sundered_ versions of the names in the class body: `_start_`, `_init_`, `_settings_`.

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'  # 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 aenum import unique
>>> @unique
... class Mistake(Enum):
...   _order_ = 'one two three'  # only needed in 2.x
...   one = 1
...   two = 2
...   three = 3
...   four = 3
Traceback (most recent call last):
...
ValueError: duplicate names found in <aenum '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:

```
>>> [n for n, mbr in Shape.__members__.items() if mbr.name != n]
['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 consider using an OrderedEnum.

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 Enum 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 Enum subclassing

A new *Enum* class must have one base Enum class, up to one concrete data type, and as many *object*-based mixin classes as needed. The order of these base classes is:

```
def EnumName([mix-in, ...,] [data-type,] base-enum):
    pass
```

Also, 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: <aenum 'MoreColor'> cannot extend <aenum 'Color'>
```

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 aenum.test 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.

> ***Note***
>
> With pickle protocol version 4 (introduced in Python 3.4) it is possible to easily pickle enums nested in other classes.

Enum Functional API

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

```
>>> Animal = Enum('Animal', 'ant bee cat dog')
>>> Animal
<aenum '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 aenum 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.

IntFlag

The next variation of `Enum` provided, `IntFlag`, is also based on `int`. The difference being `IntFlag` members can be combined using the bitwise operators (&, |, ^, ~) and the result is still an `IntFlag` member. However, as the name implies, `IntFlag` members also subclass `int` and can be used wherever an `int` is used. Any operation on an `IntFlag` member besides the bit-wise operations will lose the `IntFlag` membership.

Sample `IntFlag` class:

```
>>> from aenum import IntFlag
>>> class Perm(IntFlag):
...     _order_ = 'R W X'
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
```

```
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

It is also possible to name the combinations:

```
>>> class Perm(IntFlag):
...     _order_ = 'R W X'
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm: 0>
```

Another important difference between `IntFlag` and `Enum` is that if no flags are set (the value is 0), its boolean evaluation is `False`:

```
>>> Perm.R & Perm.X
<Perm: 0>
>>> bool(Perm.R & Perm.X)
False
```

Because `IntFlag` members are also subclasses of `int` they can be combined with them:

```
>>> Perm.X | 4
<Perm.R|X: 5>
```

If the result is not a `Flag` then, depending on the `_boundary_` setting, an exception is raised (`STRICT`), the extra bits are lost (`CONFORM`), or it reverts to an int (`EJECT`):

```
>>> from aenum import STRICT, CONFORM, EJECT
>>> Perm._boundary_ = STRICT
>>> Perm.X | 8
Traceback (most recent call last):
...
ValueError: 9 is not a valid Perm
```

```
>>> Perm._boundary_ = EJECT
>>> Perm.X | 8
9
```

```
>>> Perm._boundary_ = CONFORM
>>> Perm.X | 8
<Perm.X: 1>
```

Flag

The last variation is `Flag`. Like `IntFlag`, `Flag` members can be combined using the bitwise operators (&, |, ^, ~). Unlike `IntFlag`, they cannot be combined with, nor compared against, any other `Flag` enumeration, nor `int`. While it is possible to specify the values directly it is recommended to use `auto` as the value and let `Flag` select an appropriate value.

Like `IntFlag`, if a combination of `Flag` members results in no flags being set, the boolean evaluation is `False`:

```
>>> from aenum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

Individual flags should have values that are powers of two (1, 2, 4, 8, ...), while combinations of flags won't:

```
--> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
--> Color.WHITE
<Color.WHITE: 7>
```

Giving a name to the "no flags set" condition does not change its boolean value:

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

Flags can be iterated over to retrieve the individual truthy flags in the value:

```
>>> class Color(Flag):
...     _order_ = 'BLACK RED BLUE GREEN WHITE'
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
```

```
>>> list(Color.GREEN)
[<Color.GREEN: 4>]
>>> list(Color.WHITE)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 4>]
```

> ***Note***
>
> For the majority of new code, `Enum` and `Flag` are strongly recommended, since `IntEnum` and `IntFlag` break some semantic promises of an enumeration (by being comparable to integers, and thus by transitivity to other unrelated enumerations). `IntEnum` and `IntFlag` should be used only in cases where `Enum` and `Flag` will not do; for example, when integer constants are replaced with enumerations, or for interoperability with other systems.

Others

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

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

This demonstrates how similar derived enumerations can be defined; for example a `MyStrEnum` 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 `MyIntEnum` 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 or be convertible into that type. This restriction does not apply to mix-ins which only add methods and don't specify another data type.
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 MyIntEnum) treat the enum member as its mixed-in type.
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.

> ***Note***
>
> If you override the `__str__` method, then it will be used to provide the string portion of the `format()` call.

> ***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.

Extra Goodies

aenum supports a few extra techniques not found in the stdlib version.

enum

If you have several items to initialize your `Enum` members with and would like to use keyword arguments, the `enum` helper is for you:

```
>>> from aenum import enum
>>> class Presidents(Enum):
...     Washington = enum('George Washington', circa=1776, death=1797)
...     Jackson = enum('Andrew Jackson', circa=1830, death=1837)
...     Lincoln = enum('Abraham Lincoln', circa=1860, death=1865)
...
>>> Presidents.Lincoln
<Presidents.Lincoln: enum('Abraham Lincoln', circa=1860, death=1865)>
```

extend_enum

For those rare cases when you need to create your `Enum` in pieces, you can use `extend_enum` to add new members after the initial creation (the new member is returned):

```
>>> from aenum import extend_enum
>>> class Color(Enum):
...     red = 1
...     green = 2
...     blue = 3
...
>>> list(Color)
[<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
>>> extend_enum(Color, 'opacity', 4)
<Color.opacity: 4>
>>> list(Color)
[<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>, <Color.opacity: 4>]
>>> Color.opacity in Color
True
>>> Color.opacity.name == 'opacity'
True
>>> Color.opacity.value == 4
True
>>> Color(4)
<Color.opacity: 4>
>>> Color['opacity']
<Color.opacity: 4>

--> Color.__members__
```

```
OrderedDict([
    ('red', <Color.red: 1>),
    ('green', <Color.green: 2>),
    ('blue', <Color.blue: 3>),
    ('opacity', <Color.opacity: 4>)
    ])
```

constant

If you need to have some constant value in your Enum that isn't a member, use `constant`:

```
>>> from aenum import constant
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     G = constant(6.67300E-11)
...     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)
...         return self.G * self.mass / (self.radius * self.radius)
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
>>> Planet.G
6.673e-11
>>> Planet.G = 9
Traceback (most recent call last):
...
AttributeError: Planet: cannot rebind constant 'G'
```

skip

If you need a standard attribute that is not converted into an Enum member, use `skip`:

```
>>> from aenum import skip
>>> class Color(Enum):
...     red = 1
...     green = 2
...     blue = 3
...     opacity = skip(0.45)
...
>>> Color.opacity
0.45
>>> Color.opacity = 0.77
>>> Color.opacity
0.77
```

start

`start` can be used to turn on auto-numbering (useful for when you don't care which numbers are assigned as long as they are consistent and in order) The Python 3 version can look like this:

```
>>> class Color(Enum, start=1):                # doctest: +SKIP
...     red, green, blue
...
>>> Color.blue
<Color.blue: 3>
```

This can also be done in Python 2, albeit not as elegantly (this also works in Python 3):

```
>>> class Color(Enum):                         # doctest: +SKIP
...     _start_ = 1
...     red = auto()
...     green = auto()
...     blue = auto()
...
>>> Color.blue
<Color.blue: 3>
```

init

If you need an `__init__` method that does nothing besides save its arguments, `init` is for you:

```
>>> class Planet(Enum, init='mass radius'):      # doctest: +SKIP
...     MERCURY = (3.303e+23, 2.4397e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     G = constant(6.67300E-11)
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant  (m3 kg-1 s-2)
...         return self.G * self.mass / (self.radius * self.radius)
...
>>> Planet.JUPITER.value
(1.9e+27, 71492000.0)
>>> Planet.JUPITER.mass
1.9e+27
```

> ***Note***
>
> Just as with `start` above, in Python 2 you must put the keyword as a _sunder_ in the class body -- `_init_ = 'mass radius'`.

init and missing values

If `_init_` calls for values that are not supplied, `_generate_next_value_` will be called in an effort to generate them. Here is an example in Python 2:

```
>>> from aenum import Enum
>>> class SelectionEnum(Enum):
...     _init_ = 'db user'
...     def __new__(cls, *args, **kwds):
...         count = len(cls.__members__)
...         obj = object.__new__(cls)
...         obj._count = count
...         obj._value_ = args
...         return obj
...     @staticmethod
...     def _generate_next_value_(name, start, count, values, *args, **kwds):
...         return (name, ) + args
...
>>> class NotificationType(SelectionEnum):
...     # usually, name is the same as db
...     # but not for blanks
...     blank = '', ''
...     C = 'Catalog'
...     S = 'Sheet'
...     B = 'Both'
...
>>> NotificationType.blank
<NotificationType.blank: ('', '')>
>>> NotificationType.B
<NotificationType.B: ('B', 'Both')>
>>> NotificationType.B.db
'B'
>>> NotificationType.B.user
'Both'
```

combining Flag with other data types

Flag does support being combined with other data types. To support this you need to provide a `_create_pseudo_member_values_` method which will be called with the members in a composite flag. You may also need to provide a custom `__new__` method:

```
>>> class AnsiFlag(str, Flag):
...     def __new__(cls, value, code):
...         str_value = '\x1b[%sm' % code
...         obj = str.__new__(cls, str_value)
...         obj._value_ = value
...         obj.code = code
...         return obj
...     @classmethod
...     def _create_pseudo_member_values_(cls, members, *values):
...         code = ';'.join(m.code for m in members)
...         return values + (code, )
...     _order_ = 'FG_Red FG_Green BG_Magenta BG_White'
...     FG_Red = '31'             # ESC [ 31 m      # red
...     FG_Green = '32'           # ESC [ 32 m      # green
...     BG_Magenta = '45'         # ESC [ 35 m      # magenta
...     BG_White = '47'           # ESC [ 37 m      # white
...
>>> color = AnsiFlag.BG_White | AnsiFlag.FG_Red
>>> repr(color)
```

```
'<AnsiFlag.FG_Red|BG_White: 9>'
>>> str.__repr__(color)
"'\\x1b[31;47m'"
```

> ***Note***
>
> If you do not provide your own `_create_pseudo_member_values_` the flags may still combine, but may be missing functionality.

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 <aenum '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 (the first three are included in the module), 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__* without modifying the class after it is created.

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):
...     _order_ = 'red green blue'
...     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__`).

> ***Note***
>
> `__dir__` is not changed in the Python 2 line as it messes up some of the decorators included in the stdlib.

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 are accessible as *EnumClass.member1.member2* -- but only if no other constant/property exists:

```
>>> class FieldTypes(Enum):
...     name = 1
...     value = 2
...     size = 3
...
>>> FieldTypes.size
<FieldTypes.size: 3>
>>> FieldTypes.value.size
<FieldTypes.size: 3>
>>> FieldTypes.size.value    # NOT <FieldTypes.value: 2>
3
```

The `__members__` attribute is only available on the class.

`__members__` is always an `OrderedDict`, with the order being the definition order in Python 3.x or the order in `_order_` in Python 2.7; if no `_order_` was specified in Python 2.7 then the order of `__members__` is either increasing value or alphabetically by name.

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 (in Python 3.x):

```
--> 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`.

Note

If you create your own `__new__` you should set the `_value_` in it; if you do not, aenum will try to, but will raise a `TypeError` if it cannot.

If the stdlib `enum` is available (Python 3.4+ and it hasn't been shadowed by, for example, `enum34`) then aenum will be a subclass of it.

To use the `AddValue`, `MultiValue`, `NoAlias`, and `Unique` flags in Py2 or Py2/Py3 codebases, use `_settings_ = ...` in the class body.

To use `init` in Py2 or Py2/Py3 codebases use `_init_` in the class body.

To use `start` in Py2 or Py2/Py3 codebases use `_start_` in the class body.

When creating class bodies dynamically, put any variables you need to use into `_ignore_`:

```
>>> from datetime import timedelta
>>> from aenum import NoAlias
>>> class Period(timedelta, Enum):
...     '''
...     different lengths of time
...     '''
...     _init_ = 'value period'
...     _settings_ = NoAlias
...     _ignore_ = 'Period i'
...     Period = vars()
...     for i in range(31):
...         Period['day_%d' % i] = i, 'day'
...     for i in range(15):
...         Period['week_%d' % i] = i*7, 'week'
...
>>> hasattr(Period, '_ignore_')
False
>>> hasattr(Period, 'Period')
False
>>> hasattr(Period, 'i')
False
```

The name listed in `_ignore_`, as well as `_ignore_` itself, will not be present in the final enumeration as neither attributes nor members.

Note

except for __dunder__ attributes/methods, all _sunder_ attributes must be before any thing else in the class body

> ***Note***
>
> all _sunder_ attributes that affect member creation are only looked up in the last `Enum` class listed in the class header

Creating NamedTuples

Simple

The most common way to create a new NamedTuple will be via the functional API:

```
>>> from aenum import NamedTuple
>>> Book = NamedTuple('Book', 'title author genre', module=__name__)
```

This creates a `NamedTuple` called `Book` that will always contain three items, each of which is also addressable as `title`, `author`, or `genre`.

`Book` instances can be created using positional or keyword argements or a mixture of the two:

```
>>> b1 = Book('Lord of the Rings', 'J.R.R. Tolkien', 'fantasy')
>>> b2 = Book(title='Jhereg', author='Steven Brust', genre='fantasy')
>>> b3 = Book('Empire', 'Orson Scott Card', genre='scifi')
```

If too few or too many arguments are used a `TypeError` will be raised:

```
>>> b4 = Book('Hidden Empire')
Traceback (most recent call last):
...
TypeError: values not provided for field(s): author, genre
>>> b5 = Book(genre='business')
Traceback (most recent call last):
...
TypeError: values not provided for field(s): title, author
```

As a `class` the above `Book` `NamedTuple` would look like:

```
>>> class Book(NamedTuple):
...     title = 0
...     author = 1
...     genre = 2
...
```

For compatibility with the stdlib `namedtuple`, NamedTuple also has the `_asdict`, `_make`, and `_replace` methods, and the `_fields` attribute, which all function similarly:

```
>>> class Point(NamedTuple):
...     x = 0, 'horizontal coordinate', 1
...     y = 1, 'vertical coordinate', -1
...
>>> class Color(NamedTuple):
...     r = 0, 'red component', 11
```

```
...     g = 1, 'green component', 29
...     b = 2, 'blue component', 37
...
>>> Pixel = NamedTuple('Pixel', Point+Color, module=__name__)
>>> pixel = Pixel(99, -101, 255, 128, 0)

>>> pixel._asdict()
OrderedDict([('x', 99), ('y', -101), ('r', 255), ('g', 128), ('b', 0)])

>>> Point._make((4, 5))
Point(x=4, y=5)

>>> purple = Color(127, 0, 127)
>>> mid_gray = purple._replace(g=127)
>>> mid_gray
Color(r=127, g=127, b=127)

>>> pixel._fields
['x', 'y', 'r', 'g', 'b']

>>> Pixel._fields
['x', 'y', 'r', 'g', 'b']
```

Advanced

The simple method of creating `NamedTuples` requires always specifying all possible arguments when creating instances; failure to do so will raise exceptions:

```
>>> class Point(NamedTuple):
...     x = 0
...     y = 1
...
>>> Point()
Traceback (most recent call last):
...
TypeError: values not provided for field(s): x, y
>>> Point(1)
Traceback (most recent call last):
...
TypeError: values not provided for field(s): y
>>> Point(y=2)
Traceback (most recent call last):
...
TypeError: values not provided for field(s): x
```

However, it is possible to specify both docstrings and default values when creating a `NamedTuple` using the class method:

```
>>> class Point(NamedTuple):
...     x = 0, 'horizontal coordinate', 0
...     y = 1, 'vertical coordinate', 0
...
>>> Point()
Point(x=0, y=0)
>>> Point(1)
```

```
Point(x=1, y=0)
>>> Point(y=2)
Point(x=0, y=2)
```

It is also possible to create `NamedTuples` that only have named attributes for certain fields; any fields without names can still be accessed by index:

```
>>> class Person(NamedTuple):
...     fullname = 2
...     phone = 5
...
>>> p = Person('Ethan', 'Furman', 'Ethan Furman',
...            'ethan at stoneleaf dot us',
...            'ethan.furman', '999.555.1212')
>>> p
Person('Ethan', 'Furman', 'Ethan Furman', 'ethan at stoneleaf dot us',
       'ethan.furman', '999.555.1212')
>>> p.fullname
'Ethan Furman'
>>> p.phone
'999.555.1212'
>>> p[0]
'Ethan'
```

In the above example the last named field was also the last field possible; in those cases where you don't need to have the last possible field named, you can provide a `_size_` of `TupleSize.minimum` to declare that more fields are okay:

```
>>> from aenum import TupleSize
>>> class Person(NamedTuple):
...     _size_ = TupleSize.minimum
...     first = 0
...     last = 1
...
```

or, optionally if using Python 3:

```
>>> class Person(NamedTuple, size=TupleSize.minimum):      # doctest: +SKIP
...     first = 0
...     last = 1
```

and in use:

```
>>> Person('Ethan', 'Furman')
Person(first='Ethan', last='Furman')

>>> Person('Ethan', 'Furman', 'ethan.furman')
Person('Ethan', 'Furman', 'ethan.furman')

>>> Person('Ethan', 'Furman', 'ethan.furman', 'yay Python!')
Person('Ethan', 'Furman', 'ethan.furman', 'yay Python!')

>>> Person('Ethan')
Traceback (most recent call last):
```

```
...
TypeError: values not provided for field(s): last
```

Also, for those cases where even named fields may not be present, you can specify `TupleSize.variable`:

```
>>> class Person(NamedTuple):
...     _size_ = TupleSize.variable
...     first = 0
...     last = 1
...

>>> Person('Ethan')
Person('Ethan')

>>> Person(last='Furman')
Traceback (most recent call last):
...
TypeError: values not provided for field(s): first
```

Creating new `NamedTuples` from existing `NamedTuples` is simple:

```
>>> Point = NamedTuple('Point', 'x y')
>>> Color = NamedTuple('Color', 'r g b')
>>> Pixel = NamedTuple('Pixel', Point+Color, module=__name__)
>>> Pixel
<NamedTuple 'Pixel'>
```

The existing fields in the bases classes are renumbered to fit the new class, but keep their doc strings and default values. If you use standard subclassing:

```
>>> Point = NamedTuple('Point', 'x y')
>>> class Pixel(Point):
...     r = 2, 'red component', 11
...     g = 3, 'green component', 29
...     b = 4, 'blue component', 37
...
>>> Pixel.__fields__
['x', 'y', 'r', 'g', 'b']
```

You must manage the numbering yourself.

Creating NamedConstants

A `NamedConstant` class is created much like an `Enum`:

```
>>> from aenum import NamedConstant
>>> class Konstant(NamedConstant):
...     PI = 3.14159
...     TAU = 2 * PI

>>> Konstant.PI
<Konstant.PI: 3.14159>
```

```
>> print(Konstant.PI)
3.14159

>>> Konstant.PI = 'apple'
Traceback (most recent call last):
...
AttributeError: cannot rebind constant <Konstant.PI>

>>> del Konstant.PI
Traceback (most recent call last):
...
AttributeError: cannot delete constant <Konstant.PI>
```