`enum` — Support for enumerations

New in version 3.4.

Source code: Lib/enum.py

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.

> **Note:** Case of Enum Members
>
> Because Enums are used to represent constants we recommend using UPPER_CASE names for enum members, and will be using that style in our examples.

Module Contents

This module defines four enumeration classes that can be used to define unique sets of names and values: `Enum`, `IntEnum`, `Flag`, and `IntFlag`. It also defines one decorator, `unique()`, and one helper, `auto`.

class `enum.`**`Enum`**

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

class `enum.`**`IntEnum`**

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

class `enum.`**`IntFlag`**

Base class for creating enumerated constants that can be combined using the bitwise operators without losing their `IntFlag` membership. `IntFlag` members are also subclasses of `int`.

class `enum.`**`Flag`**

Base class for creating enumerated constants that can be combined using the bitwise operations without losing their `Flag` membership.

`enum.`**`unique`**()

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

class `enum.`**`auto`**

Instances are replaced with an appropriate value for Enum members. By default, the initial value starts at 1.

New in version 3.6: `Flag`, `IntFlag`, `auto`

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: Enum member values

Member values can be anything: `int`, `str`, etc.. If the exact value is unimportant you may use `auto` instances and an appropriate value will be chosen for you. Care must be taken if you mix `auto` with other values.

Note: Nomenclature

- The class `Color` is an *enumeration* (or *enum*)
- The attributes `Color.RED`, `Color.GREEN`, etc., are *enumeration members* (or *enum members*) and are functionally constants.
- 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 definition order:

```
>>> class Shake(Enum):
...     VANILLA = 7
...     CHOCOLATE = 4
...     COOKIES = 9
...     MINT = 3
...
>>> for shake in Shake:
...     print(shake)
...
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
```

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 you 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 with the same name is invalid:

```
>>> 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):
...     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>
```

Note: Attempting to create a member with the same name as an already defined attribute (another member, a method, etc.) or attempting to create an attribute with the same name as a member is not allowed.

Ensuring unique enumeration values

By default, enumerations allow multiple names as aliases for the same value. When this behavior isn't desired, the following decorator can be used to ensure each value is used only once in the enumeration:

`@enum.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:

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

Using automatic values

If the exact value is unimportant you can use `auto`:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

The values are chosen by `_generate_next_value_()`, which can be overridden:

```
>>> class AutoName(Enum):
```

```
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAS
```

Note: The goal of the default `_generate_next_value_()` methods is to provide the next `int` in sequence with the last `int` provided, but the way it does this is an implementation detail and may change.

Note: The `_generate_next_value_()` method must be defined before any members.

Iteration

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 read-only ordered mapping of names to members. It includes all names defined in the enumeration, including the aliases:

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_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 !
['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: '<' not supported between instances of 'Color' and 'Color'
```

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: names that start and end 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 special methods (`__str__()`, `__add__()`, etc.), descriptors (methods are also descriptors), and variable names listed in `_ignore_`.

Note: if your enumeration defines `__new__()` and/or `__init__()` then any value(s) 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:

```
class 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: 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 test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
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 it is possible to easily pickle enums nested in other classes.

It is possible to modify how Enum members are pickled/unpickled by defining `__reduce_ex__()` in the enumeration class.

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 (use the `start` parameter to specify a different starting value). A new class derived from `Enum` is returned. In other words, the above assignment to `Animal` is equivalent to:

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

The reason for defaulting to `1` as the starting number and not `0` is that `0` is `False` in a boolean sense, but enum members all evaluate to `True`.

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:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

Warning: If `module` is not supplied, and Enum cannot determine what it is, the new Enum members will not be unpicklable; to keep errors closer to the source, pickling will be disabled.

The new pickle protocol 4 also, in some circumstances, relies on `__qualname__` being set to the location where pickle will be able to find the class. For example, if the class was made available in class SomeData in the global scope:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='SomeData.Animal
```

The complete signature is:

```
Enum(value='NewEnumName', names=<...>, *, module='...', qualname='...',
```

value: What the new Enum class will record as its name.

names: The Enum members. This can be a whitespace or comma separated string (values will start at 1 unless otherwise specified):

```
'RED GREEN BLUE' | 'RED,GREEN,BLUE' | 'RED, GREEN, BLUE'
```

or an iterator of names:

```
['RED', 'GREEN', 'BLUE']
```

or an iterator of (name, value) pairs:

```
[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]
```

or a mapping:

```
{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}
```

module: name of module where new Enum class can be found.

qualname: where in module new Enum class can be found.

type: type to mix in to new Enum class.

start: number to start counting at if only names are passed in.

Changed in version 3.5: The *start* parameter was added.

Derived Enumerations

IntEnum

The first variation of `Enum` that is provided 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]
```

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.

New in version 3.6.

Sample `IntFlag` class:

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     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):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm.-8: -8>
```

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

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

```
>>> Perm.X | 8
<Perm.8|X: 9>
```

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.

New in version 3.6.

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

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

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 `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 type.
3. When another data type is mixed in, the `value` attribute is *not the same* as the enum member itself, although it is equivalent and will compare equal.
4. %-style formatting: *%s* and *%r* call the `Enum` class's `__str__()` and `__repr__()` respectively; other codes (such as *%i* or *%h* for IntEnum) treat the enum member as its mixed-in type.
5. Formatted string literals, `str.format()`, and `format()` will use the mixed-in type's `__format__()` unless `__str__()` or `__format__()` is overridden in the subclass, in which case the overridden methods or `Enum` methods will be used. Use the !s and !r format codes to force usage of the `Enum` class's `__str__()` and `__repr__()` methods.

When to use `__new__()` vs. `__init__()`

`__new__()` must be used whenever you want to customize the actual value of the `Enum` member. Any other modifications may go in either `__new__()` or `__init__()`, with `__init__()` being preferred.

For example, if you want to pass several items to the constructor, but only want one of them to be the value:

```
>>> class Coordinate(bytes, Enum):
...     """
```

```
...     Coordinate with binary codes that can be indexed by the int code
...     """
...     def __new__(cls, value, label, unit):
...         obj = bytes.__new__(cls, [value])
...         obj._value_ = value
...         obj.label = label
...         obj.unit = unit
...         return obj
...     PX = (0, 'P.X', 'km')
...     PY = (1, 'P.Y', 'km')
...     VX = (2, 'V.X', 'km/s')
...     VY = (3, 'V.Y', 'km/s')
...

>>> print(Coordinate['PY'])
Coordinate.PY

>>> print(Coordinate(3))
Coordinate.VY
```

Interesting examples

While `Enum`, `IntEnum`, `IntFlag`, and `Flag` 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.

Omitting values

In many use-cases one doesn't care what the actual value of an enumeration is. There are several ways to define this type of simple enumeration:

- use instances of `auto` for the value
- use instances of `object` as the value
- use a descriptive string as the value
- use a tuple as the value and a custom `__new__()` to replace the tuple with an `int` value

Using any of these methods signifies to the user that these values are not important, and also enables one to add, remove, or reorder members without having to renumber the remaining members.

Whichever method you choose, you should provide a `repr()` that also hides the (unimportant) value:

```
>>> class NoValue(Enum):
...     def __repr__(self):
```

```
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
...
```

Using `auto`

Using `auto` would look like:

```
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

Using `object`

Using `object` would look like:

```
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

Using a descriptive string

Using a string as the value would look like:

```
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
'go'
```

Using a custom `__new__()`

Using an auto-numbering `__new__()` would look like:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
```

```
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
2
```

To make a more general purpose `AutoNumber`, add `*args` to the signature:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls, *args):       # this is the only change from abc
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
```

Then when you inherit from `AutoNumber` you can write your own `__init__` to handle any extra arguments:

```
>>> class Swatch(AutoNumber):
...     def __init__(self, pantone='unknown'):
...         self.pantone = pantone
...     AUBURN = '3497'
...     SEA_GREEN = '1246'
...     BLEACHED_CORAL = () # New color, no Pantone code yet!
...
>>> Swatch.SEA_GREEN
<Swatch.SEA_GREEN: 2>
>>> Swatch.SEA_GREEN.pantone
'1246'
>>> Swatch.BLEACHED_CORAL.pantone
'unknown'
```

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.

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):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True
```

DuplicateFreeEnum

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

```
>>> class DuplicateFreeEnum(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 DuplicateFreeEnum:  %r --> %
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
```

```
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE' --> 'GREE
```

Note: This is a useful example for subclassing Enum to add or change other behaviors as well as disallowing aliases. If the only desired change is disallowing aliases, the `unique()` decorator can be used instead.

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
```

TimePeriod

An example to show the `_ignore_` attribute in use:

```
>>> from datetime import timedelta
>>> class Period(timedelta, Enum):
...     "different lengths of time"
...     _ignore_ = 'Period i'
```

```
...     Period = vars()
...     for i in range(367):
...         Period['day_%d' % i] = i
...
>>> list(Period)[:2]
[<Period.day_0: datetime.timedelta(0)>, <Period.day_1: datetime.timedelt
>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>, <Period.day_366: dateti
```

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_enum_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

Supported `__dunder__` names

`__members__` is a read-only ordered mapping of `member_name:member` items. It is only available on the class.

`__new__()`, if specified, must create and return the enum members; it is also a very good idea to set the member's `_value_` appropriately. Once all the members are created it is no longer used.

Supported `_sunder_` names

- `_name_` – name of the member

- `_value_` – value of the member; can be set / modified in `__new__`
- `_missing_` – a lookup function used when a value is not found; may be overridden
- `_ignore_` – a list of names, either as a `list` or a `str`, that will not be transformed into members, and will be removed from the final class
- `_order_` – used in Python 2/3 code to ensure member order is consistent (class attribute, removed during class creation)
- `_generate_next_value_` – used by the Functional API and by `auto` to get an appropriate value for an enum member; may be overridden

New in version 3.6: `_missing_`, `_order_`, `_generate_next_value_`

New in version 3.7: `_ignore_`

To help keep Python 2 / Python 3 code in sync an `_order_` attribute can be provided. It will be checked against the actual order of the enumeration and raise an error if the two do not match:

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_
```

Note: In Python 2 code the `_order_` attribute is necessary as definition order is lost before it can be recorded.

_Private__names

Private names will be normal attributes in Python 3.10 instead of either an error or a member (depending on if the name ends with an underscore). Using these names in 3.9 will issue a `DeprecationWarning`.

`Enum` member type

`Enum` members are instances of their `Enum` class, and are normally accessed as `EnumClass.member`. Under certain circumstances they can also be accessed as `EnumClass.member.member`, but you should never do this as that lookup may fail or, worse, return something besides the `Enum` member you are looking for (this is another good reason to use all-uppercase names for members):

```
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

Changed in version 3.5.

Boolean value of `Enum` classes and members

`Enum` members that are mixed with non-`Enum` types (such as `int`, `str`, etc.) are evaluated according to the mixed-in type's rules; otherwise, all members evaluate as `True`. To make your own Enum's boolean evaluation depend on the member's value add the following to your class:

```
def __bool__(self):
    return bool(self.value)
```

`Enum` classes always evaluate as `True`.

`Enum` classes with methods

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', '
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value
```

Combining members of `Flag`

If a combination of Flag members is not named, the `repr()` will include all named flags and all named combinations of flags that are in the value:

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
```

```
...     CYAN = GREEN | BLUE
...
>>> Color(3)  # named combination
<Color.YELLOW: 3>
>>> Color(7)      # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>
```