Enum 指南

`Enum` 是一组绑定到唯一值的符号名称。 它们类似于全局变量，但提供了更好用的 `repr()`、分组、类型安全和一些其他特性。

它们最适用于当某个变量可选的值有限时。例如，从一周中选取一天：

```
>>> from enum import Enum
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
```

或是 RGB 三原色：

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
```

正如你所见，创建一个 `Enum` 就是简单地写一个继承 `Enum` 的类。

> **备注：** 枚举成员名的大小写
>
> 由于枚举被用来代表常量，并有助于避免混入类方法/属性和枚举名之间发生名称冲突，我们强烈建议用大写形式的名称表示成员，我们将在我们的示例中使用此风格。

根据枚举的性质，某个成员的值可能不一定用得上，但无论如何都能用那个值构造对应的成员：

```
>>> Weekday(3)
<Weekday.WEDNESDAY: 3>
```

如你所见，成员的 `repr()` 会显示枚举名称、成员名称和值。 成员的 `str()` 只会显示枚举名称和成员名称:

```
>>> print(Weekday.THURSDAY)
Weekday.THURSDAY
```

枚举成员的 *类型* 就是其所属的枚举:

```
>>> type(Weekday.MONDAY)
<enum 'Weekday'>
```

```
>>> isinstance(Weekday.FRIDAY, Weekday)
True
```

枚举成员带有一个只包含了它们的 `name` 的属性:

```
>>> print(Weekday.TUESDAY.name)
TUESDAY
```

类似地，它们还有一个包含其 `value` 的属性:

```
>>> Weekday.WEDNESDAY.value
3
```

不同于许多只把枚举当作名称/值对的的语言，Python 枚举还可以添加行为。 例如，`datetime.date` 有两个方法用来返回星期序号: `weekday()` 和 `isoweekday()`。 两者的区别在于一个是以 0-6 计数而另一个是以 1-7。 这一点无须我们自己来记住而是可以向 `Weekday` 枚举添加一个方法用来从 `date` 实例提取日期并返回匹配的枚举成员:

```
@classmethod
def from_date(cls, date):
    return cls(date.isoweekday())
```

完整的 `Weekday` 枚举现在看起来是这样的:

```
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...     #
...     @classmethod
...     def from_date(cls, date):
...         return cls(date.isoweekday())
```

现在可以知道今天是星期几了:

```
>>> from datetime import date
>>> Weekday.from_date(date.today())
<Weekday.TUESDAY: 2>
```

当然，如果换个日子读到这篇文章，应该看到当天是周几。

如果我们的变量只需记录一天那么这个 `Weekday` 枚举很好用，但是如果我们需要记录好几天呢？ 可能我们要写一个函数来描述一星期内的家务，并且不想使用 `list` —— 我们可以使用另一种类型的 `Enum`:

```
>>> from enum import Flag
>>> class Weekday(Flag):
...     MONDAY = 1
```

```
...     TUESDAY = 2
...     WEDNESDAY = 4
...     THURSDAY = 8
...     FRIDAY = 16
...     SATURDAY = 32
...     SUNDAY = 64
```

这里做了两处改动：继承了 `Flag`，而且值都是2的幂。

就像上面的原始 `Weekday` 枚举一样，我们可以有单个选择:

```
>>> first_week_day = Weekday.MONDAY
>>> first_week_day
<Weekday.MONDAY: 1>
```

但 `Flag` 也允许将几个成员并入一个变量:

```
>>> weekend = Weekday.SATURDAY | Weekday.SUNDAY
>>> weekend
<Weekday.SATURDAY|SUNDAY: 96>
```

甚至可以在一个 `Flag` 变量上进行迭代:

```
>>> for day in weekend:
...     print(day)
Weekday.SATURDAY
Weekday.SUNDAY
```

好吧，让我们来安排家务吧:

```
>>> chores_for_ethan = {
...     'feed the cat': Weekday.MONDAY | Weekday.WEDNESDAY | Weekday.FRIDAY,
...     'do the dishes': Weekday.TUESDAY | Weekday.THURSDAY,
...     'answer SO questions': Weekday.SATURDAY,
...     }
```

一个显示某天家务的函数:

```
>>> def show_chores(chores, day):
...     for chore, days in chores.items():
...         if day in days:
...             print(chore)
...
>>> show_chores(chores_for_ethan, Weekday.SATURDAY)
answer SO questions
```

对于成员的实际取值无关紧要的情况，你可以省事地使用 `auto()` 来设置值:

```
>>> from enum import auto
>>> class Weekday(Flag):
...     MONDAY = auto()
...     TUESDAY = auto()
...     WEDNESDAY = auto()
...     THURSDAY = auto()
```

```
...     FRIDAY = auto()
...     SATURDAY = auto()
...     SUNDAY = auto()
...     WEEKEND = SATURDAY | SUNDAY
```

枚举成员及其属性的编程访问

有时，要在程序中访问枚举成员（如，开发时不知道颜色的确切值，`Color.RED` 不适用的情况）。`Enum` 支持如下访问方式:

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

若要用 *名称* 访问枚举成员时，可使用枚举项:

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

如果你有一个枚举成员并需要它的 `name` 或 `value`:

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

重复的枚举成员和值

两个枚举成员的名称不能相同:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: 'SQUARE' already defined as 2
```

然而，一个枚举成员可以关联多个其他名称。如果两个枚举项 `A` 和 `B` 具有相同值（并且首先定义的是 `A`），则 `B` 是成员 `A` 的别名。对 `A` 按值检索将会返回成员 `A`。按名称检索 `B` 也会返回成员 `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>
```

备注： 不允许创建与已定义属性（其他成员、方法等）同名的成员，也不支持创建与现有成员同名的属性。

确保枚举值唯一

默认情况下，枚举允许多个名称作为同一个值的别名。若不想如此，可以使用 `unique()` 装饰器:

```
>>> 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
```

使用自动设定的值

如果具体的枚举值无所谓是什么，可以使用 `auto`:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> [member.value for member in Color]
[1, 2, 3]
```

枚举值由 `_generate_next_value_()` 来选取，它可以被重写:

```
>>> class AutoName(Enum):
...     @staticmethod
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> [member.value for member in Ordinal]
['NORTH', 'SOUTH', 'EAST', 'WEST']
```

备注： `_generate_next_value_()` 方法必须在任何成员之前定义。

迭代遍历

对枚举成员的迭代遍历不会列出别名:

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
>>> list(Weekday)
[<Weekday.MONDAY: 1>, <Weekday.TUESDAY: 2>, <Weekday.WEDNESDAY: 4>, <Weekday.THURS
```

请注意 `Shape.ALIAS_FOR_SQUARE` 和 `Weekday.WEEKEND` 等别名不会被显示。

特殊属性 `__members__` 是一个名称与成员间的只读有序映射。包含了枚举中定义的所有名称，包括别名:

```
>>> 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>)
```

`__members__` 属性可用于获取枚举成员的详细信息。比如查找所有别名:

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

备注： 旗标的别名包括带有多个旗标设置的值，如 `3`，以及不设置任何旗标，即 `0`。

比较运算

枚举成员是按 ID 进行比较的:

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

枚举值之间无法进行有序的比较。枚举的成员不是整数（另请参阅下文 IntEnum）：

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and 'Color'
```

相等性比较的定义如下:

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

与非枚举值的比较将总是不等的（同样 `IntEnum` 有意设计为其他的做法，参见下文）：

```
>>> Color.BLUE == 2
False
```

警告： 重载模块是可能的 -- 如果载入的模块包含枚举，它们将被重新创建，而新成员的标识号/相等性比较不一定会通过。

合法的枚举成员和属性

以上大多数示例都用了整数作为枚举值。使用整数确实简短方便（并且是 Functional API 默认提供的值），但并非强制要求。绝大多数情况下，人们并不关心枚举的实际值是什么。但如果值确实重要，可以使用任何值。

枚举是 Python 的类，可带有普通方法和特殊方法。假设有如下枚举:

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # 在这里 self 是成员
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # 在这里 cls 是枚举
...         return cls.HAPPY
...
```

那么:

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

对于允许内容的规则如下：以单下划线开始和结尾的名称是由枚举保留而不可使用的；在枚举中定义的其他属性将成为该枚举的成员，例外项则包括特殊方法 (`__str__()`, `__add__()`, etc.)、描述器 (方法也属于描述器) 以及在 `_ignore_` 中列出的变量名。

注意：如果你的枚举定义了 `__new__()` 和/或 `__init__()`，则给予枚举成员的任何值都将被传递给这些方法。 参见 Planet 中的示例。

备注： 如果定义了 `__new__()` 方法，它会在创建 Enum 成员期间被使用；随后它将被 Enum 的 `__new__()` 所替换，该方法会在类创建后被用于查找现有成员。 详情参见 何时应使用 __new__() 或 __init__()。

受限的 Enum 子类化

新建的 `Enum` 类必须包含：一个枚举基类、至多一种数据类型和按需提供的基于 `object` 的混合类。这些基类的顺序如下:

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

仅当未定义任何成员时，枚举类才允许被子类化。因此不得有以下写法:

```
>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: <enum 'MoreColor'> cannot extend <enum 'Color'>
```

但以下代码是可以的:

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

如果定义了成员的枚举也能被子类化，则类型与实例的某些重要不可变规则将会被破坏。另一方面，一组枚举类共享某些操作也是合理的。（请参阅例程 OrderedEnum ）

数据类支持

当从 `dataclass` 继承时，`__repr__()` 将忽略被继承类的名称。 例如:

```
>>> from dataclasses import dataclass, field
>>> @dataclass
... class CreatureDataMixin:
...     size: str
...     legs: int
...     tail: bool = field(repr=False, default=True)
...
>>> class Creature(CreatureDataMixin, Enum):
...     BEETLE = 'small', 6
```

```
...     DOG = 'medium', 4
...
>>> Creature.DOG
<Creature.DOG: size='medium', legs=4>
```

使用 `dataclass()` 参数 `repr=False` 来使用标准的 `repr()`。

> *在 3.12 版本发生变更:* 只有数据类字段会被显示在值区域中，数据类名称不会被显示。

备注: 向 `Enum` 及其子类添加 `dataclass()` 装饰器是不受支持的。 它不会引发任何错误，但会在运行时产生非常怪异的结果，例如不同的成员彼此相等:

```
>>> @dataclass                # 不要这样做：没有任何意义
... class Color(Enum):
...     RED = 1
...     BLUE = 2
...
>>> Color.RED is Color.BLUE
False
>>> Color.RED == Color.BLUE  # 问题在这里：它们不应该相等
True
```

打包（pickle）

枚举类型可以被打包和解包:

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

打包的常规限制同样适用于枚举类型：必须在模块的最高层级定义，因为解包操作要求可从该模块导入。

备注: 用 pickle 协议版本 4 可以轻松地将嵌入其他类中的枚举进行打包。

通过在枚举类中定义 `__reduce_ex__()`，可以修改枚举成员的 pickled/unpicled 方式。默认方法是根据值进行的，但具有复杂值的枚举可能需要根据名称进行:

```
>>> import enum
>>> class MyEnum(enum.Enum):
...     __reduce_ex__ = enum.pickle_by_enum_name
```

备注: 不建议为旗标使用基于名称的方式，因为未命名的别名将无法解封。

函数式 API

`Enum` 类可调用并提供了以下函数式 API：

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>, <Animal.DOG: 4>]
```

该 API 的语义类似于 `namedtuple`。调用 `Enum` 的第一个参数是枚举的名称。

第二个参数是枚举成员名称的 *来源*。它可以是以空白分隔的名称字符串、名称序列、包含键/值对的 2 元组序列或名称到值的映射（如字典）。后两个选项可为枚举指定任意值；其他选项则自动指定从 1 开始的递增整数（使用 `start` 参数可指定不同的起始值）。返回值是一个从 `Enum` 派生的新类。也就是说，上述对 `Animal` 的赋值相当于：

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

默认从 `1` 开始而非 `0`，因为 `0` 是布尔值 `False`，但默认的枚举成员都被视作 `True`。

对使用函数式 API 创建的枚举进行封存，可能会很棘手，因为要使用栈帧的实现细节来尝试找出枚举是在哪个模块中创建的（例如当你使用了另一个模块中的实用函数时它就可能失败，在 IronPython 或 Jython 上也可能无效）。解决办法是像下面这样显式地指定模块名称：

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

警告： 如果未提供 `module`，且 Enum 无法确定是哪个模块，新的 Enum 成员将不可被解封；为了让错误尽量靠近源头，封存将被禁用。

在某些情况下，新的 pickle 版本 4 协议还需要 `__qualname__` 在 pickle 能够找到类的位置。 例如，如果该类是在全局作用域内的 SomeData 类中可见:

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

完整的签名为:

```
Enum(
    value='NewEnumName',
    names=<...>,
    *,
    module='...',
    qualname='...',
    type=<mixed-in class>,
    start=1,
    )
```

- *value*: 新枚举类将会作为其名称记录的值。

- *names*: 枚举的成员。 这可以是一个用空格或逗号分隔的字符串（值将从 1 开始除非另外指定）：

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

  或是一个名称的迭代器对象:

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

  或是一个 (名称, 值) 对的迭代器对象:

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

  或是一个映射对象:

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

- *module*: 新枚举类所在的模块名。
- *qualname*: 新枚举类在模块内的位置。
- *type*: 要混入到新枚举类的类型。
- *start*: 当只传入名称时要使用的起始计数编号。

> *在 3.5 版本发生变更:* 增加了 *start* 形参。

派生的枚举

IntEnum

所提供的第一个变种 `Enum` 同时也是 `int` 的一个子类。 `IntEnum` 的成员可与整数进行比较；通过扩展，不同类型的整数枚举也可以相互进行比较:

```
>>> 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
```

不过，它们仍然不可与标准 `Enum` 枚举进行比较:

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
```

```
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

IntEnum 值在其他方面的行为都如你预期的一样类似于整数:

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

StrEnum

所提供的第二种 Enum 变体同时也是 str 的一个子类。 StrEnum 的成员可与字符串进行比较；通过扩展，不同类型的字符串枚举也可以相互进行比较。

> *Added in version 3.11.*

IntFlag

所提供的下一种 Enum 变体 IntFlag 也是基于 int 的。 不同之处在于 IntFlag 成员可以用位运算符 (&, |, ^, ~) 进行组合并且如果可能的话其结果仍将是 IntFlag 成员。 与 IntEnum 类似，IntFlag 成员也是整数并且可以用于任何使用 int 的地方。

> **备注:** 除位操作外，其他所有对 IntFlag 成员的操作，都会失去 IntFlag 成员资格。
>
> 导致 IntFlag 值无效的位操作将失去 IntFlag 成员资格。详见 FlagBoundary。

> *Added in version 3.6.*

> *在 3.11 版本发生变更.*

示例 IntFlag 类:

```
>>> 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
```

对于组合同样可以进行命名:

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

备注: 命名的枚举组合被视作别名。别名在迭代过程中不会显示，但可以通过值查询返回。

在 3.11 版本发生变更.

IntFlag 和 Enum 的另一个重要区别在于如果没有设置任何旗标（值为 0），则其布尔值为 False:

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

因为 IntFlag 成员也是 int 的子类，他们可以相互组合（但可能会失去 IntFlag 成员资格:

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

>>> Perm.X + 8
9
```

备注: 否运算符 `~`，始终会返回一个 IntFlag 成员的正值:

```
>>> (~Perm.X).value == (Perm.R|Perm.W).value == 6
True
```

IntFlag 成员也可被迭代遍历:

```
>>> list(RW)
[<Perm.R: 4>, <Perm.W: 2>]
```

Added in version 3.11.

标志

最后一个变体是 `Flag`。与 `IntFlag` 类似，`Flag` 成员可用按位运算符 (&, |, ^, ~) 组合。与 `IntFlag` 不同的是，它们不可与其它 `Flag` 枚举或 `int` 进行组合或比较。 虽然可以直接指定值，但推荐使用 `auto` 作为值来让 `Flag` 选择适当的值。

> *Added in version 3.6.*

与 `IntFlag` 类似，如果 `Flag` 成员的某种组合导致没有设置任何旗标，则其布尔值为 `False`:

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

单个旗标的值应当为二的乘方 (1, 2, 4, 8, ...)，而旗标的组合则无此限制:

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

对 "no flags set" 条件指定一个名称并不会改变其布尔值:

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

`Flag` 成员也可被迭代遍历:

```
>>> purple = Color.RED | Color.BLUE
>>> list(purple)
[<Color.RED: 1>, <Color.BLUE: 2>]
```

> *Added in version 3.11.*

备注： 对于大多数新代码，强烈推荐使用 `Enum` 和 `Flag`，因为 `IntEnum` 和 `IntFlag` 打破了枚举的某些语义约定（例如可以同整数进行比较，并因而导致此行为被传递给其他无关的枚举）。

`IntEnum` 和 `IntFlag` 的使用应当仅限于 `Enum` 和 `Flag` 无法使用的场合；例如，当使用枚举替代整数常量时，或是与其他系统进行交互操作时。

其他事项

虽然 `IntEnum` 是 `enum` 模块的一部分，但要独立实现也应该相当容易:

```
class IntEnum(int, ReprEnum):   # 或用 Enum 而不是 ReprEnum
    pass
```

这里演示了类似的派生枚举可以如何被定义；例如，一个混入了 `float` 而不是 `int` 的 `FloatEnum`。

几条规则：

1. 当子类化 `Enum` 时，混入类型必须出现在基类序列中的 `Enum` 类本身之前，如以上 `IntEnum` 的例子所示。
2. 混入类型必须是可子类化的。 例如，`bool` 和 `range` 是不可子类化的因而如果被用作混入类型就将在枚举创建期间抛出错误。
3. 虽然 `Enum` 可以拥有任意类型的成员，不过一旦你混合了附加类型，则所有成员必须为相应类型的值，如在上面的例子中即为 `int`。 此限制不适用于仅添加方法而未指定另一数据类型的混合类。
4. 当混合了另一数据类型时，`value` 属性将 *不同于* 枚举成员本身，不过它们仍会保持等价且比较结果也相等。
5. `data type` 是一个定义了 `__new__()` 的混入对象，或者是一个 `dataclass`
6. % 形式的格式化： `%s` 和 `%r` 会分别调用 `Enum` 类的 `__str__()` 和 `__repr__()`；其他代码（如 `%i` 或 `%h` 用于 IntEnum）会将枚举成员当作对应的混入类型。
7. 格式化字符串字面值, `str.format()` 和 `format()` 将使用枚举的 `__str__()` 方法。

备注: 由于 `IntEnum`, `IntFlag` 和 `StrEnum` 被设计为现有常量的无缝替换，它们的 `__str__()` 方法已被重置为其数据类型的 `__str__()` 方法。

何时应使用 `__new__()` 或 `__init__()`

当你想要定制 `Enum` 成员的实际值时你必须使用 `__new__()`。 任何其他修改则可使用 `__new__()` 或 `__init__()`，其中 `__init__()` 更为推荐。

举例来说，如果你要向构造器传入多个条目，但只希望将其中一个作为值:

```
>>> 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
```

警告: *不要* 调用 `super().__new__()`，因为只能找到仅用于查找的 `__new__`；请改为直接使用该数据类型。

细节要点

支持的 __dunder__ 名称

`__members__` 是一个由 `member_name:member` 条目组成的只读有序映射。 它只在类上可用。

如果指定了 `__new__()`，它必须创建并返回枚举成员；相应地设置成员的 `_value_` 也是一个很好的主意。 一旦所有成员都创建完成它将不再被使用。

支持的 _sunder_ 名称

- `_name_` -- 成员的名称
- `_value_` -- 成员的值；可在 `__new__` 中设置
- `_missing_()` -- 当未找到某个值时所使用的查找函数；可被重写
- `_ignore_` -- 一个名称列表，可以为 `list` 或 `str`，它不会被转化为成员，并将从最终类中移除
- `_generate_next_value_()` -- 用于为枚举成员获取适当的值；可被重写
- `_add_alias_()` -- 添加一个新名称作为现有成员的别名。
- `_add_value_alias_()` -- 添加一个新值作为现有成员的别名。 参见 MultiValueEnum 中的示例。

  备注: 对于标准的 `Enum` 类来说下一个被选择的值将是已有的最高值加一。

  对于 `Flag` 类来说下一个选择的值将是下一个最高的二的幂数。

  > *在 3.13 版本发生变更:* 在之前版本中将会使用最近的值而不是最高的值。

> *Added in version 3.6:* `_missing_`, `_order_`, `_generate_next_value_`

> *Added in version 3.7:* `_ignore_`

> *Added in version 3.13:* `_add_alias_`, `_add_value_alias_`

用于帮助 Python 2 / Python 3 代码保持同步以便提供 _order_ 属性。 它将与枚举的实际顺序进行检查并会在两者不匹配时引发错误:

```
>>> 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_:
  ['RED', 'BLUE', 'GREEN']
  ['RED', 'GREEN', 'BLUE']
```

备注: 在 Python 2 代码中 _order_ 属性是必须的，因为定义顺序在被记录之前就已丢失。

_Private__names

私有名称 不会被转换为枚举成员，而是保持为普通属性。

> *在 3.11 版本发生变更.*

Enum 成员类型

枚举成员是其枚举类的实例，并且通常以 EnumClass.member 的形式来访问。 在特定场景下，如编写自定义枚举行为，可直接从一个成员访问另一个成员的能力是很有用的，并且是受支持的；但是，为了避免成员名与混入类属性/方法之间发生名称冲突，强烈建议使用大写形式的名称。

> *在 3.5 版本发生变更.*

创建与其他数据类型混合的成员

当使用 Enum 来子类化其他数据类型，如 int 或 str 时，所有在 = 之后的值都会被传递给该数据类型的构造器。 例如:

```
>>> class MyEnum(IntEnum):       # help(int) -> int(x, base=10) -> integer
...     example = '11', 16       # 这样 x='11' 而 base=16
...
>>> MyEnum.example.value         # 而 hex(11) 为...
17
```

Enum 类和成员的布尔值

与非 Enum 类型（如 int、str 等）混合的枚举类会根据混合类型的规则进行计算；否则，所有成员都计算为 True。为了使你自己的枚举的布尔值取决于成员的值，请在你的类中添加以下内容:

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

普通的 `Enum` 类总是计算为 `True`。

带有方法的 Enum 类

如果你给你的枚举子类提供了额外的方法，如下面的 Planet 类那样，这些方法将显示在成员的，而不是类的 `dir()` 中:

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '_
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'mass', 'name', 'radius', 'surface_gravity'
```

组合 Flag 的成员

遍历 `Flag` 成员的组合将只返回由一个比特组成的成员:

```
>>> 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.RED|GREEN|BLUE: 7>
```

Flag 和 IntFlag 的细节

使用以下代码段作为我们的例子:

```
>>> class Color(IntFlag):
...     BLACK = 0
...     RED = 1
...     GREEN = 2
...     BLUE = 4
...     PURPLE = RED | BLUE
...     WHITE = RED | GREEN | BLUE
...
```

下列情况为True:

- 单比特标志是典型的
- 多比特和零比特标志是别名
- 迭代过程中只返回典型的标志:

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

- 取负一个标志或标志集会返回一个新的标志/标志集和其对应的正整数值:

```
>>> Color.BLUE
<Color.BLUE: 4>

>>> ~Color.BLUE
<Color.RED|GREEN: 3>
```

- 伪标志的名称是由其成员的名称构建的:

```
>>> (Color.RED | Color.GREEN).name
'RED|GREEN'

>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> (Perm.R & Perm.W).name is None  # effectively Perm(0)
True
```

- 多位标志，又称别名，可以从操作中返回:

```
>>> Color.RED | Color.BLUE
<Color.PURPLE: 5>

>>> Color(7)  # or Color(-1)
<Color.WHITE: 7>

>>> Color(0)
<Color.BLACK: 0>
```

- 成员 / 包含检测：零值旗标总是会被视为包含:

```
>>> Color.BLACK in Color.WHITE
True
```

  在其他情况下，仅当一个旗标的所有比特位都包含于另一个旗标中才会返回 True:

```
>>> Color.PURPLE in Color.WHITE
True

>>> Color.GREEN in Color.PURPLE
False
```

有一个新的边界机制，控制如何处理超出范围的/无效的比特: `STRICT`, `CONFORM`, `EJECT`, `KEEP`。

- STRICT --> 当出现无效的值时，会触发一个异常。
- CONFORM --> 丢弃任何无效的比特
- EJECT --> 失去Flag的状态，成为一个普通的int，其值为给定值。
- KEEP --> 保留额外的比特
  - 保留Flag状态和额外的比特
  - 额外的比特不会在迭代中显示出来

- 在repr()和str()中确实显示了额外的比特

默认的标志为 `STRICT`，`IntFlag` 默认为 `EJECT`，`_convert_` 默认为 `KEEP` (需要 `KEEP` 的例子见 `ssl.Options`)。

枚举和旗标有何差异?

Enum有一个自定义的元类，它影响到派生的 `Enum` 类和它们的实例（成员）的许多方面。

枚举类

`EnumType` 元类负责提供 `__contains__()`, `__dir__()`, `__iter__()` 及其他方法来允许人们在 `Enum` 类上做一些在常规类上会失败的事情，比如 `list(Color)` 或 `some_enum_var in Color`。`EnumType` 负责确保最终的 `Enum` 类上的各种其他方法是正确的（比如 `__new__()`, `__getnewargs__()`, `__str__()` 和 `__repr__()` 等）。

旗标类

旗标具有扩展的别名视图：为了符合规范，旗标的值必须为二的乘方，且名称不可重复。 因此，除了别名的定义 `Enum` 之外，没有值 (即 `0`) 或是几个二的乘方值之和 (如 `3`) 的旗标也会被视为别名。

枚举成员（即实例）

有关枚举成员的最有趣的一点在于它们都是单例。 `EnumType` 会在创建枚举类本身时全部创建它们，然后放置一个自定义的 `__new__()` 以通过只返回现有的成员实例来确保没有新的成员被实例化。

旗标成员

旗标成员可以如 `Flag` 类一样被迭代，并且只有规范的成员会被返回。 例如:

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

（请注意 `BLACK`, `PURPLE` 和 `WHITE` 将不显示。）

对一个旗标成员取反将返回对应的正值，而不是负值 --- 例如:

```
>>> ~Color.RED
<Color.GREEN|BLUE: 6>
```

旗标成员具有与它们所包含的二的乘方值的数量相对应的长度。 例如:

```
>>> len(Color.PURPLE)
2
```

枚举指导手册

虽然 Enum, IntEnum, StrEnum, Flag 和 IntFlag 有望能涵盖大多数的使用情况，但它们不能涵盖所有情况。 这里有一些不同类型的枚举的方法，可以直接使用，或者作为创建定制枚举的范例。

省略值

在许多应用场景中，人们并不关心枚举的实际值是什么。 有几种方式可用来定义这种类型的简单枚举:

- 使用 auto 的实例作为值
- 使用 object 的实例作为值
- 使用描述性的字符串作为值
- 使用一个元组作为值并用自定义的 __new__() 以一个 int 值来替代该元组

使用以上任何一种方法均可向用户指明值并不重要，并且使人能够添加、移除或重排序成员而不必改变其余成员的数值。

使用 auto

使用 auto 的形式如下:

```
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN: 3>
```

使用 object

使用 object 的形式如下:

```
>>> class Color(Enum):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN: <object object at 0x...>>
```

这也是一个可以说明为什么你会需要编写自己的 __repr__() 的好例子:

```
>>> class Color(Enum):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...     def __repr__(self):
...         return "<%s.%s>" % (self.__class__.__name__, self._name_)
...
>>> Color.GREEN
<Color.GREEN>
```

使用描述性字符串

使用字符串作为值的形式如下:

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

使用自定义的 __new__()

使用自动编号的 __new__() 看起来会是这样:

```
>>> class AutoNumber(Enum):
...     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: 2>
```

要实现更通用的 `AutoNumber`，请添加 `*args` 到签名中:

```
>>> class AutoNumber(Enum):
...     def __new__(cls, *args):      # 这是相比上面的唯一改变
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
```

这样当你从 `AutoNumber` 继承时你将可以编写你自己的 `__init__` 来处理任何附加参数:

```
>>> class Swatch(AutoNumber):
...     def __init__(self, pantone='unknown'):
...         self.pantone = pantone
...     AUBURN = '3497'
...     SEA_GREEN = '1246'
...     BLEACHED_CORAL = () # 新的颜色，还没有 Pantone 代码!
...
>>> Swatch.SEA_GREEN
<Swatch.SEA_GREEN: 2>
>>> Swatch.SEA_GREEN.pantone
'1246'
```

```
>>> Swatch.BLEACHED_CORAL.pantone
'unknown'
```

备注： 如果定义了 `__new__()` 方法，它会在创建 Enum 成员期间被使用；随后它将被 Enum 的 `__new__()` 所替换，该方法会在类创建后被用来查找现有成员。

警告： *不要* 调用 `super().__new__()`，因为只能找到仅用于查找的 `__new__`；请改为直接使用该数据类型 -- 例如:

```
obj = int.__new__(cls, value)
```

OrderedEnum

一个有序枚举，它不是基于 `IntEnum`，因此保持了正常的 `Enum` 不变特性（例如不可与其他枚举进行比较）：

```
>>> 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

如果发现重复的成员名称则会引发一个错误而不是创建一个别名:

```
>>> 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 --> %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' --> 'GREEN'
```

备注： 这个例子适用于子类化 Enum 来添加或改变禁用别名以及其他行为。 如果需要的改变只是禁用别名，也可以选择使用 `unique()` 装饰器。

MultiValueEnum

支持每个成员有多个值:

```
>>> class MultiValueEnum(Enum):
...     def __new__(cls, value, *values):
...         self = object.__new__(cls)
...         self._value_ = value
...         for v in values:
...             self._add_value_alias_(v)
...         return self
...
>>> class DType(MultiValueEnum):
...     float32 = 'f', 8
...     double64 = 'd', 9
...
>>> DType('f')
<DType.float32: 'f'>
>>> DType(9)
<DType.double64: 'd'>
```

Planet

如果定义了 `__new__()` 或 `__init__()`，则枚举成员的值将被传给这些方法:

```
>>> 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

一个演示如何使用 _ignore_ 属性的例子:

```
>>> 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.timedelta(days=1)>
>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>, <Period.day_366: datetime.timedel
```

子类化 EnumType

虽然大多数枚举需求可以通过自定义 Enum 子类来满足，无论是用类装饰器还是自定义函数，EnumType 可以被子类化以提供不同的枚举体验。