HOWTO Enum

Release 3.12.2

Guido van Rossum and the Python development team

fevereiro 07, 2024

Python Software Foundation
Email: docs@python.org

Sumário

Uma classe `Enum` é um conjunto de nomes simbólicos vinculados a valores únicos. Eles são semelhantes às variáveis globais, mas oferecem um `repr()` mais útil, agrupamento, segurança de tipo e alguns outros recursos.

Eles são mais úteis quando você tem uma variável que pode ter uma seleção limitada de valores. Por exemplo, os dias da semana:

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

Ou talvez as cores primárias RGB:

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

Como você pode ver, criar um `Enum` é tão simples quanto escrever uma classe que herda do próprio `Enum`.

Nota: Caso de membros de Enums

Como os Enums são usados para representar constantes, e para ajudar a evitar problemas com nomes conflitando entre métodos/atributos de classes mixin e nomes enum, nós fortemente recomendamos o uso de nomes em UPPER_CASE(em caixa alta) para membros, e usaremos esse estilo em nossos exemplos.

Dependendo da natureza do enum, o valor de um membro pode ou não ser importante, mas de qualquer forma esse valor pode ser usado para obter o membro correspondente:

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

Como você pode ver, o `repr()` de um membro mostra o nome do enum, o nome do membro e o valor. O `str()` de um membro mostra apenas o nome do enum e o nome do membro:

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

O *tipo* de um membro de enumeração é o enum ao qual ele pertence:

```
>>> type(Weekday.MONDAY)
<enum 'Weekday'>
>>> isinstance(Weekday.FRIDAY, Weekday)
True
```

Os membros do Enum têm um atributo que contém apenas seu `name`:

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

Da mesma forma, eles têm um atributo para seu `value`:

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

Ao contrário de muitas linguagens que tratam enumerações apenas como pares de nome/valor, Enums do Python podem ter comportamento adicionado. Por exemplo, `datetime.date` tem dois métodos para retornar o dia da semana: `weekday()` e `isoweekday()`. A diferença é que um deles conta de 0 a 6 e o outro de 1 a 7. Em vez de acompanhar isso nós mesmos, podemos adicionar um método ao enum de `Weekday` para extrair o dia da instância de `date` e retornar o membro enum correspondente:

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

O enum de `Weekday` completa agora se parece com isso:

```
>>> 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())
```

Agora podemos descobrir o que é hoje! Observar:

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

Claro, se você estiver lendo isso em algum outro dia, você verá esse dia.

Este enum `Weekday` é ótimo se nossa variável precisar apenas de um dia, mas e se precisarmos de vários? Talvez estejamos escrevendo uma função para traçar tarefas durante uma semana e não queremos usar uma `list` – poderíamos usar um tipo diferente de `Enum`:

```
>>> from enum import Flag
>>> class Weekday(Flag):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 4
...     THURSDAY = 8
...     FRIDAY = 16
...     SATURDAY = 32
...     SUNDAY = 64
```

Nós mudamos duas coisas: estamos herdando de `Flag`, e os valores são todos potências de 2.

Assim como o enum `Weekday` original acima, podemos ter uma seleção única:

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

Porem `Flag` também nos permite combinar vários membros em uma única variável:

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

Você pode até mesmo iterar sobre uma variável `Flag`:

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

Certo, vamos configurar algumas tarefas domésticas:

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

E a função para mostrar as tarefas domésticas para um determinado dia:

```
>>> 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
```

Nos casos onde os valores reais dos membros não importam, você pode economizar trabalho e usar `auto()` para os valores:

```
>>> from enum import auto
>>> class Weekday(Flag):
...     MONDAY = auto()
...     TUESDAY = auto()
...     WEDNESDAY = auto()
...     THURSDAY = auto()
...     FRIDAY = auto()
...     SATURDAY = auto()
...     SUNDAY = auto()
...     WEEKEND = SATURDAY | SUNDAY
```

1 Acesso programático aos membros da enumeração e seus atributos.

Em alguns momentos, é util ter acesso aos membros na enumeração de forma programática(ou seja, em situações em que `Color.RED` não é adequado porque a cor exata não é conhecida no momento da escrita do programa).“Enum“ permite esse tipo de acesso:

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

(continua na próxima página)

(continuação da página anterior)

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

Se você deseja ter acesso aos membros do enum pelo *nome*, use o acesso por itens:

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

Se você tem um membro do enum e precisa do seu `name` ou `value`:

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

2 Duplicar membros do enum e seus valores.

Ter dois membros de um enum com o mesmo nome é inválido:

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

Porém, um membro do enum pode ter outros nomes associados a ele. Dado dois membros `A` e `B` com o mesmo valor (e `A` definido primeiro), `B` é um apelido para o membro `A`. A pesquisa por valor de `A` retorna o membro `A`. A Pesquisa por nome de `A` também retorna o membro `A`. A pesquisa por nome de `B` também retorna o membro `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>
```

Nota: Tentar criar um membro com o mesmo nome de um atributo já definido (outro membro, um método, etc.) ou tentar criar um atributo com o mesmo nome de um membro não é permitido.

---

3 Garantindo valores únicos de enumeração

Por padrão, enumerações permitem múltiplos nomes como apelidos para o mesmo valor. Quando esse comportamento não é desejado, você pode usar o decorador `unique()`:

```
>>> 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
```

4 Usando valores automáticos

Se o exato valor não é importante, você pode usar `auto`:

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

Os valores são escolhidos por `_generate_next_value_()`, o qual pode ser substituído:

```
>>> 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']
```

Nota: O método `_generate_next_value_()` deve ser definido antes de qualquer membro.

---

5 Iteração

Iterar sobre os membros de um enum não fornece os apelidos:

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
>>> list(Weekday)
[<Weekday.MONDAY: 1>, <Weekday.TUESDAY: 2>, <Weekday.WEDNESDAY: 4>, <Weekday.
↪THURSDAY: 8>, <Weekday.FRIDAY: 16>, <Weekday.SATURDAY: 32>, <Weekday.SUNDAY: 64>]
```

Note que os apelidos `Shape.ALIAS_FOR_SQUARE` e `Weekday.WEEKEND` não são mostrados.

O atributo especial `__members__` é um mapeamento ordenado somente leitura de nomes para os membros. Isso inclui todos os nomes definidos na enumeração, incluindo os apelidos:

```
>>> 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>)
```

O atributo `__members__` pode ser usado para um acesso programático detalhado aos membros da enumeração. Por exemplo, achar todos os apelidos:

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

Nota: Aliases for flags include values with multiple flags set, such as `3`, and no flags set, i.e. `0`.

---

6 Comparações

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
```

Aviso: It is possible to reload modules – if a reloaded module contains enums, they will be recreated, and the new members may not compare identical/equal to the original members.

7 Allowed members and attributes of enumerations

Most of 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__()`, any value(s) given to the enum member will be passed into those methods. See *Planet* for an example.

Nota: 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. See *When to use __new__() vs. __init__()* for more details.

---

8 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: <enum 'MoreColor'> cannot extend <enum '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.)

9 Dataclass support

When inheriting from a `dataclass`, the `__repr__()` omits the inherited class' name. For example:

```
>>> 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>
```

Use the `dataclass()` argument `repr=False` to use the standard `repr()`.

Alterado na versão 3.12: Only the dataclass fields are shown in the value area, not the dataclass' name.

10 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.

Nota: 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. The default method is by-value, but enums with complicated values may want to use by-name:

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

Nota: Using by-name for flags is not recommended, as unnamed aliases will not unpickle.

---

11 API funcional

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>
>>> 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 by default 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 a 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__)
```

Aviso: 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='...',
    type=<mixed-in class>,
    start=1,
    )
```

- *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.

Alterado na versão 3.5: The *start* parameter was added.

12 Derived Enumerations

12.1 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]
```

12.2 StrEnum

The second variation of `Enum` that is provided is also a subclass of `str`. Members of a `StrEnum` can be compared to strings; by extension, string enumerations of different types can also be compared to each other.

Novo na versão 3.11.

12.3 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, if possible. Like `IntEnum`, `IntFlag` members are also integers and can be used wherever an `int` is used.

Nota: Any operation on an `IntFlag` member besides the bit-wise operations will lose the `IntFlag` membership.

Bit-wise operations that result in invalid `IntFlag` values will lose the `IntFlag` membership. See `FlagBoundary` for details.

Novo na versão 3.6.

Alterado na versão 3.11.

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: 0>
>>> Perm(7)
<Perm.RWX: 7>
```

Nota: Named combinations are considered aliases. Aliases do not show up during iteration, but can be returned from by-value lookups.

Alterado na versão 3.11.

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 (but may lose `IntFlag` membership:

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

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

Nota: The negation operator, ~, always returns an `IntFlag` member with a positive value:

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

`IntFlag` members can also be iterated over:

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

Novo na versão 3.11.

12.4 Sinalizador

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.

Novo na versão 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>
>>> bool(Color.RED & Color.GREEN)
False
```

Individual flags should have values that are powers of two (1, 2, 4, 8, …), while combinations of flags will not:

```
>>> 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()
...
```

(continua na próxima página)

(continuação da página anterior)

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

`Flag` members can also be iterated over:

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

Novo na versão 3.11.

Nota: 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.

12.5 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 `FloatEnum` that mixes in `float` 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. Mix-in types must be subclassable. For example, `bool` and `range` are not subclassable and will throw an error during Enum creation if used as the mix-in type.
3. 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.
4. 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.
5. A `data type` is a mixin that defines `__new__()`, or a `dataclass`
6. %-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.
7. Formatted string literals, `str.format()`, and `format()` will use the enum's `__str__()` method.

Nota: Because `IntEnum`, `IntFlag`, and `StrEnum` are designed to be drop-in replacements for existing constants, their `__str__()` method has been reset to their data types' `__str__()` method.

---

13 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
```

Aviso: *Do not* call `super().__new__()`, as the lookup-only `__new__` is the one that is found; instead, use the data type directly.

13.1 Finer Points

Nomes `__dunder__` suportados

`__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.

Nomes `_sunder_` suportados

- `_name_` – nome do membro
- `_value_` – valor do membro; pode ser definido / modificado em `__new__`
- `_missing_` – uma função de pesquisa usada quando um valor não é encontrado; pode ser substituída
- `_ignore_` – uma lista de nomes, seja como `list` ou `str`, que não serão transformados em membros e serão removidos da classe final
- `_order_` – usado no código Python 2/3 para garantir que a ordem dos membros seja consistente (atributo de classe, removido durante a criação da classe)
- `_generate_next_value_` – used by the *Functional API* and by `auto` to get an appropriate value for an enum member; may be overridden

Nota: Para classes `Enum` padrão, o próximo valor escolhido é o último valor visto incrementado em um.

Para as classes `Flag` o próximo valor escolhido será a próxima potência de dois mais alta, independentemente do último valor visto.

Novo na versão 3.6: `_missing_`, `_order_`, `_generate_next_value_`

Novo na versão 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_:
  ['RED', 'BLUE', 'GREEN']
  ['RED', 'GREEN', 'BLUE']
```

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

_Private__names

Private names are not converted to enum members, but remain normal attributes.

Alterado na versão 3.11.

`Enum` member type

Enum members are instances of their enum class, and are normally accessed as `EnumClass.member`. In certain situations, such as writing custom enum behavior, being able to access one member directly from another is useful, and is supported; however, in order to avoid name clashes between member names and attributes/methods from mixed-in classes, upper-case names are strongly recommended.

Alterado na versão 3.5.

Creating members that are mixed with other data types

When subclassing other data types, such as `int` or `str`, with an `Enum`, all values after the = are passed to that data type's constructor. For example:

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

Boolean value of `Enum` classes and members

Enum classes 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)
```

Plain `Enum` classes always evaluate as `True`.

`Enum` classes with methods

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

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

Combining members of `Flag`

Iterating over a combination of `Flag` members will only return the members that are comprised of a single bit:

```
>>> 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` and `IntFlag` minutia

Using the following snippet for our examples:

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

the following are true:

- single-bit flags are canonical
- multi-bit and zero-bit flags are aliases

- only canonical flags are returned during iteration:

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

- negating a flag or flag set returns a new flag/flag set with the corresponding positive integer value:

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

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

- names of pseudo-flags are constructed from their members' names:

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

- multi-bit flags, aka aliases, can be returned from operations:

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

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

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

- membership / containment checking: zero-valued flags are always considered to be contained:

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

  otherwise, only if all bits of one flag are in the other flag will True be returned:

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

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

There is a new boundary mechanism that controls how out-of-range / invalid bits are handled: `STRICT`, `CONFORM`, `EJECT`, and `KEEP`:

- STRICT –> raises an exception when presented with invalid values
- CONFORM –> discards any invalid bits
- EJECT –> lose Flag status and become a normal int with the given value
- KEEP –> keep the extra bits
  - keeps Flag status and extra bits
  - extra bits do not show up in iteration
  - extra bits do show up in repr() and str()

The default for Flag is `STRICT`, the default for `IntFlag` is `EJECT`, and the default for `_convert_` is `KEEP` (see `ssl.Options` for an example of when `KEEP` is needed).

14 How are Enums and Flags different?

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

14.1 Enum Classes

The `EnumType` 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`. `EnumType` is responsible for ensuring that various other methods on the final `Enum` class are correct (such as `__new__()`, `__getnewargs__()`, `__str__()` and `__repr__()`).

14.2 Flag Classes

Flags have an expanded view of aliasing: to be canonical, the value of a flag needs to be a power-of-two value, and not a duplicate name. So, in addition to the `Enum` definition of alias, a flag with no value (a.k.a. `0`) or with more than one power-of-two value (e.g. `3`) is considered an alias.

14.3 Enum Members (aka instances)

The most interesting thing about enum members is that they are singletons. `EnumType` 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.

14.4 Flag Members

Flag members can be iterated over just like the `Flag` class, and only the canonical members will be returned. For example:

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

(Note that `BLACK`, `PURPLE`, and `WHITE` do not show up.)

Inverting a flag member returns the corresponding positive value, rather than a negative value — for example:

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

Flag members have a length corresponding to the number of power-of-two values they contain. For example:

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

15 Enum Cookbook

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

15.1 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.

Using `auto`

Using `auto` would look like:

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

Using `object`

Using `object` would look like:

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

This is also a good example of why you might want to write your own `__repr__()`:

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

Using a descriptive string

Using a string as the value would look like:

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

Usando um `__new__()` personalizado

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

```
>>> 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>
```

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

```
>>> class AutoNumber(Enum):
...     def __new__(cls, *args):      # this is the only change from above
...         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'
```

Nota: 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.

Aviso: *Do not* call `super().__new__()`, as the lookup-only `__new__` is the one that is found; instead, use the data type directly – e.g.:

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

15.2 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
```

15.3 DuplicateFreeEnum

Raises an error if a duplicate member value 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 --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
```

(continua na próxima página)

(continuação da página anterior)

```
Traceback (most recent call last):
  ...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE' --> 'GREEN'
```

Nota: 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.

15.4 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
```

15.5 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.timedelta(days=1)>]
>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>, <Period.day_366: datetime.
↪timedelta(days=366)>]
```

16 Subclassing EnumType

While most enum needs can be met by customizing `Enum` subclasses, either with class decorators or custom functions, `EnumType` can be subclassed to provide a different Enum experience.