HOWTO - Enum

Versión 3.12.3

Guido van Rossum and the Python development team

mayo 06, 2024

Python Software Foundation
Email: docs@python.org

Índice general

Un `Enum` es un conjunto de nombres simbólicos vinculados a valores únicos. Son similares a las variables globales, pero ofrecen un `repr()` más útil, agrupación, seguridad de tipos y algunas otras características.

Son más útiles cuando tiene una variable que puede tomar uno de una selección limitada de valores. Por ejemplo, los días de la semana:

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

O quizás los colores primarios RGB:

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

Como puede ver, crear un `Enum` es tan simple como escribir una clase que herede del propio `Enum`.

Nota: Caso de miembros de Enum

Dado que las enumeraciones se utilizan para representar constantes y para evitar problemas con conflictos de nombres entre los métodos/atributos de las clases mixin y los nombres de las enumeraciones, recomendamos encarecidamente utilizar nombres en MAYÚSCULAS para los miembros, y utilizaremos ese estilo en nuestros ejemplos.

Dependiendo de la naturaleza de la enumeración, el valor de un miembro puede o no ser importante, pero de cualquier manera ese valor puede usarse para obtener el miembro correspondiente:

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

Como puede ver, el `repr()` de un miembro muestra el nombre de enumeración, el nombre del miembro y el valor. El `str()` de un miembro muestra solo el nombre de enumeración y el nombre del miembro:

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

El *type* de un miembro de la enumeración es la enumeración a la que pertenece:

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

Los miembros de enumeración tienen un atributo que contiene solo su `name`:

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

Asimismo, tienen un atributo para su `value`:

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

A diferencia de muchos lenguajes que tratan las enumeraciones únicamente como pares de nombre/valor, Python Enums puede tener un comportamiento agregado. Por ejemplo, `datetime.date` tiene dos métodos para retornar el día de la semana: `weekday()` y `isoweekday()`. La diferencia es que uno de ellos cuenta de 0 a 6 y el otro de 1 a 7. En lugar de hacer un seguimiento de eso nosotros mismos, podemos agregar un método a la enumeración `Weekday` para extraer el día de la instancia `date` y retornar el miembro de enumeración coincidente:

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

La enumeración `Weekday` completa ahora se ve así:

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

¡Ahora podemos averiguar qué día de la semana es hoy! Observe:

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

Por supuesto, si estás leyendo esto en otro día, verás ese día en su lugar.

Esta enumeración `Weekday` es excelente si nuestra variable solo necesita un día, pero ¿y si necesitamos varios? Tal vez estamos escribiendo una función para trazar tareas durante una semana y no queremos usar un `list`; podríamos usar un tipo diferente de `Enum`:

```
>>> from enum import Flag
>>> class Weekday(Flag):
...     MONDAY = 1
...     TUESDAY = 2
```

(continúe en la próxima página)

(proviene de la página anterior)

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

Hemos cambiado dos cosas: somos heredados de `Flag` y los valores son todos potencia de 2.

Al igual que la enumeración `Weekday` original anterior, podemos tener una sola selección:

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

Pero `Flag` también nos permite combinar varios miembros en una sola variable:

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

Incluso puede iterar sobre una variable `Flag`:

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

Bien, preparemos algunas tareas:

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

Y una función para mostrar las tareas de un día determinado:

```
>>> 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
```

En los casos en que los valores reales de los miembros no importen, puede ahorrarse algo de trabajo y usar `auto()` para los valores:

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

(continúe en la próxima página)

(proviene de la página anterior)

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

1 Acceso programático a los miembros de la enumeración y sus atributos

A veces es útil acceder a los miembros en las enumeraciones programáticamente (es decir, situaciones en las que `Color.RED` no funcionará porque no se conoce el color exacto en el momento de escribir el programa). `Enum` permite dicho acceso:

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

Si desea acceder a los miembros de la enumeración por *name*, use el acceso a elementos:

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

Si tiene un miembro de enumeración y necesita su `name` o `value`:

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

2 Duplicar miembros y valores de enumeración

Tener dos miembros de enumeración con el mismo nombre no es válido:

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

Sin embargo, un miembro de enumeración puede tener otros nombres asociados. Dadas dos entradas `A` y `B` con el mismo valor (y `A` definido primero), `B` es un alias para el miembro `A`. La búsqueda por valor del valor de `A` retornará el miembro `A`. La búsqueda por nombre de `A` retornará el miembro `A`. La búsqueda por nombre de `B` también retornará el miembro `A`:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
```

(continúe en la próxima página)

(proviene de la página anterior)

```
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

Nota: No está permitido intentar crear un miembro con el mismo nombre que un atributo ya definido (otro miembro, un método, etc.) o intentar crear un atributo con el mismo nombre que un miembro.

---

3 Garantizar valores de enumeración únicos

De forma predeterminada, las enumeraciones permiten múltiples nombres como alias para el mismo valor. Cuando no se desea este comportamiento, puede usar el 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 Uso de valores automáticos

Si el valor exacto no es importante, puede 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]
```

Los valores son elegidos por `_generate_next_value_()`, que se pueden anular:

```
>>> class AutoName(Enum):
...     @staticmethod
...     def _generate_next_value_(name, start, count, last_values):
...         return name
```

(continúe en la próxima página)

(proviene de la página anterior)

```
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> [member.value for member in Ordinal]
['NORTH', 'SOUTH', 'EAST', 'WEST']
```

Nota: El método `_generate_next_value_()` debe definirse antes que cualquier miembro.

5 Iteración

Iterar sobre los miembros de una enumeración no proporciona los alias:

```
>>> 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 los alias `Shape.ALIAS_FOR_SQUARE` y `Weekday.WEEKEND` no se muestran.

El atributo especial `__members__` es una asignación ordenada de solo lectura de nombres a miembros. Incluye todos los nombres definidos en la enumeración, incluidos los alias:

```
>>> 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>)
```

El atributo `__members__` se puede utilizar para el acceso programático detallado a los miembros de la enumeración. Por ejemplo, encontrar todos los alias:

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

Nota: Los alias para las banderas incluyen valores con múltiples banderas establecidas, como `3`, y ningún conjunto de banderas, es decir, `0`.

6 Comparaciones

Los miembros de la enumeración se comparan por identidad:

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

Las comparaciones ordenadas entre valores de enumeración son compatibles con *not*. Los miembros de la enumeración no son números enteros (pero consulte *IntEnum* a continuación):

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

Las comparaciones de igualdad se definen aunque:

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

Las comparaciones con valores que no son de enumeración siempre comparan no iguales (nuevamente, `IntEnum` se diseñó explícitamente para comportarse de manera diferente, consulte a continuación):

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

Advertencia: Es posible recargar módulos; si un módulo recargado contiene enumeraciones, estas se crearán de nuevo y los nuevos miembros pueden no compararse como idénticos/iguales a los miembros originales.

7 Miembros permitidos y atributos de enumeraciones

La mayoría de los ejemplos anteriores usan números enteros para los valores de enumeración. El uso de números enteros es corto y práctico (y proporcionado por defecto por el *Functional API*), pero no se aplica estrictamente. En la gran mayoría de los casos de uso, a uno no le importa cuál es el valor real de una enumeración. Pero si el valor *is* es importante, las enumeraciones pueden tener valores arbitrarios.

Las enumeraciones son clases de Python y pueden tener métodos y métodos especiales como de costumbre. Si tenemos esta enumeración:

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
```

(continúe en la próxima página)

(proviene de la página anterior)

```
...         # 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
...
```

Después:

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

Las reglas para lo que está permitido son las siguientes: los nombres que comienzan y terminan con un solo guión bajo están reservados por enumeración y no se pueden usar; todos los demás atributos definidos dentro de una enumeración se convertirán en miembros de esta enumeración, con la excepción de métodos especiales (`__str__()`, `__add__()`, etc.), descriptores (los métodos también son descriptores) y nombres de variables enumerados en `_ignore_`.

Nota: si su enumeración define `__new__()` y/o `__init__()`, cualquier valor(es) dado(s) al miembro de la enumeración se pasará a esos métodos. Consulte *Planet* para ver un ejemplo.

Nota: El método `__new__()`, si está definido, se usa durante la creación de los miembros de Enum; luego se reemplaza por `__new__()` de Enum, que se usa después de la creación de clases para buscar miembros existentes. Consulte *Cuándo usar __new__() frente a __init__()* para obtener más detalles.

---

8 Subclases de Enum restringidas

Una nueva clase `Enum` debe tener una clase de enumeración base, hasta un tipo de datos concreto y tantas clases mixtas basadas en `object` como sea necesario. El orden de estas clases base es:

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

Además, la subclasificación de una enumeración solo se permite si la enumeración no define ningún miembro. Así que esto está prohibido:

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

Pero esto está permitido:

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

Permitir la subclasificación de enumeraciones que definen miembros conduciría a una violación de algunas invariantes importantes de tipos e instancias. Por otro lado, tiene sentido permitir compartir algún comportamiento común entre un grupo de enumeraciones. (Consulte *OrderedEnum* para ver un ejemplo).

9 Soporte de Dataclass

Cuando se hereda de una `dataclass`, el `__repr__()` omite el nombre de la clase heredada. Por ejemplo:

```
>>> 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>
```

Utilice el argumento `repr=False` de `dataclass()` para utilizar el `repr()` estándar.

Distinto en la versión 3.12: Solo se muestran los campos de la dataclass en el área de valores, no el nombre de la dataclass.

10 Serialización (Pickling)

Las enumeraciones se pueden serializar y deserializar:

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

Se aplican las restricciones habituales para el *pickling*: las enumeraciones serializables deben definirse en el nivel superior de un módulo, ya que el decapado requiere que se puedan importar desde ese módulo.

Nota: Con la versión 4 del protocolo pickle es posible deserializar fácilmente enumeraciones anidadas en otras clases.

Es posible modificar la forma en que los miembros de la enumeración se serialicen / deserialicen definiendo `__reduce_ex__()` en la clase de enumeración. El método predeterminado es por valor, pero las enumeraciones con valores complicados pueden querer utilizar por nombre:

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

Nota: No se recomienda usar banderas por nombre , ya que los alias sin nombre no se desempaquetarán.

11 API funcional

Se puede llamar a la clase `Enum`, que proporciona la siguiente API funcional:

```
>>> 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>]
```

La semántica de esta API se asemeja a `namedtuple`. El primer argumento de la llamada a `Enum` es el nombre de la enumeración.

El segundo argumento es el *source* de nombres de miembros de enumeración. Puede ser una cadena de nombres separados por espacios en blanco, una secuencia de nombres, una secuencia de 2 tuplas con pares clave/valor o una asignación (por ejemplo, un diccionario) de nombres a valores. Las dos últimas opciones permiten asignar valores arbitrarios a las enumeraciones; los otros asignan automáticamente números enteros crecientes que comienzan con 1 (use el parámetro `start` para especificar un valor inicial diferente). Se retorna una nueva clase derivada de `Enum`. En otras palabras, la asignación anterior a `Animal` es equivalente a:

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

La razón por la que se toma por defecto `1` como el número inicial y no `0` es que `0` es `False` en un sentido booleano, pero por defecto todos los miembros de la enumeración se evalúan como `True`.

Deserializar las enumeraciones creadas con la API funcional puede ser complicado, ya que los detalles de implementación de la pila de marcos se usan para tratar de averiguar en qué módulo se está creando la enumeración (por ejemplo, fallará si usa una función de utilidad en un módulo separado, y también puede no trabajar en IronPython o Jython). La solución es especificar el nombre del módulo explícitamente de la siguiente manera:

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

Advertencia: Si no se proporciona `module` y Enum no puede determinar de qué se trata, los nuevos miembros de Enum no serán seleccionables; para mantener los errores más cerca de la fuente, se desactivará el decapado.

El nuevo protocolo pickle 4 también, en algunas circunstancias, depende de que `__qualname__` se establezca en la ubicación donde pickle podrá encontrar la clase. Por ejemplo, si la clase estuvo disponible en la clase SomeData en el ámbito global:

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

La firma completa es:

```
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'
  ```

  o un iterador de nombres:

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

  o un iterador de (nombre, valor) pares:

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

  o un mapeo:

  ```
  {'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.

Distinto en la versión 3.5: Se agregó el parámetro *start*.

12 Enumeraciones derivadas

12.1 IntEnum

La primera variación de `Enum` que se proporciona también es una subclase de `int`. Los miembros de un `IntEnum` se pueden comparar con números enteros; por extensión, las enumeraciones enteras de diferentes tipos también se pueden comparar entre sí:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

Sin embargo, aún no se pueden comparar con las enumeraciones `Enum` estándar:

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

Los valores `IntEnum` se comportan como números enteros en otras formas que esperaría:

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

12.2 StrEnum

La segunda variación de `Enum` que se proporciona también es una subclase de `str`. Los miembros de un `StrEnum` se pueden comparar con cadenas; por extensión, las enumeraciones de cadenas de diferentes tipos también se pueden comparar entre sí.

Added in version 3.11.

12.3 IntFlag

La siguiente variación de `Enum` proporcionada, `IntFlag`, también se basa en `int`. La diferencia es que los miembros `IntFlag` se pueden combinar usando los operadores bit a bit (&, |, ^, ~) y el resultado sigue siendo un miembro `IntFlag`, si es posible. Al igual que `IntEnum`, los miembros `IntFlag` también son números enteros y se pueden utilizar siempre que se utilice un `int`.

Nota: Cualquier operación en un miembro `IntFlag` además de las operaciones bit a bit perderá la pertenencia a `IntFlag`.

Las operaciones bit a bit que den como resultado valores `IntFlag` no válidos perderán la pertenencia a `IntFlag`. Ver `FlagBoundary` para más detalles.

Added in version 3.6.

Distinto en la versión 3.11.

Ejemplo de clase `IntFlag`:

```
>>> 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
```

También es posible nombrar las combinaciones:

```
>>> 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: Las combinaciones con nombre se consideran alias. Los alias no aparecen durante la iteración, pero se pueden devolver a partir de búsquedas por valor.

Distinto en la versión 3.11.

Otra diferencia importante entre `IntFlag` y `Enum` es que si no se establecen banderas (el valor es 0), su evaluación booleana es `False`:

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

Debido a que los miembros `IntFlag` también son subclases de `int`, se pueden combinar con ellos (pero pueden perder la membresía `IntFlag`:

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

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

Nota: El operador de negación, ~, siempre retorna un miembro `IntFlag` con un valor positivo:

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

Los miembros `IntFlag` también se pueden iterar sobre:

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

Added in version 3.11.

12.4 Bandera

La última variación es `Flag`. Al igual que `IntFlag`, los miembros de `Flag` se pueden combinar mediante los operadores bit a bit (&, |, ^, ~). A diferencia de `IntFlag`, no se pueden combinar ni comparar con ninguna otra enumeración `Flag` ni con `int`. Si bien es posible especificar los valores directamente, se recomienda usar `auto` como valor y dejar que `Flag` seleccione un valor apropiado.

Added in version 3.6.

Al igual que `IntFlag`, si una combinación de miembros `Flag` da como resultado que no se establezcan indicadores, la evaluación booleana es `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
```

Las banderas individuales deben tener valores que sean potencias de dos (1, 2, 4, 8, …), mientras que las combinaciones de banderas no:

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

Dar un nombre a la condición «sin banderas establecidas» no cambia su valor booleano:

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

(continúe en la próxima página)

(proviene de la página anterior)

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

Los miembros `Flag` también se pueden iterar sobre:

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

Added in version 3.11.

Nota: Para la mayoría del código nuevo, se recomienda encarecidamente `Enum` y `Flag`, ya que `IntEnum` y `IntFlag` rompen algunas promesas semánticas de una enumeración (al ser comparables con los números enteros y, por lo tanto, por la transitividad a otras enumeraciones no relacionadas). `IntEnum` y `IntFlag` deben usarse solo en los casos en que `Enum` y `Flag` no sirvan; por ejemplo, cuando las constantes enteras se reemplazan con enumeraciones, o para la interoperabilidad con otros sistemas.

12.5 Otros

Si bien `IntEnum` es parte del módulo `enum`, sería muy simple de implementar de forma independiente:

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

Esto demuestra cómo se pueden definir enumeraciones derivadas similares; por ejemplo, un `FloatEnum` que se mezcla en `float` en lugar de `int`.

Algunas reglas:

1. Al subclasificar `Enum`, los tipos de combinación deben aparecer antes que `Enum` en la secuencia de bases, como en el ejemplo anterior de `IntEnum`.
2. Los tipos mixtos deben ser subclasificables. Por ejemplo, `bool` y `range` no son subclasificables y generarán un error durante la creación de Enum si se usan como tipo de combinación.
3. Si bien `Enum` puede tener miembros de cualquier tipo, una vez que mezcle un tipo adicional, todos los miembros deben tener valores de ese tipo, p. `int` anterior. Esta restricción no se aplica a los complementos que solo agregan métodos y no especifican otro tipo.
4. Cuando se mezcla otro tipo de datos, el atributo `value` es *not the same* como el propio miembro de la enumeración, aunque es equivalente y se comparará igual.
5. Un `data type` es un mixin que define `__new__()`, o una `dataclass`
6. Formato de estilo %: `%s` y `%r` llaman a `__str__()` y `__repr__()` de la clase `Enum` respectivamente; otros códigos (como `%i` o `%h` para IntEnum) tratan el miembro de enumeración como su tipo mixto.
7. Formatted string literals, `str.format()` y `format()` usarán el método `__str__()` de la enumeración.

Nota: Dado que `IntEnum`, `IntFlag` y `StrEnum` están diseñados para ser reemplazos directos de constantes existentes, su método `__str__()` se ha restablecido al método `__str__()` de sus tipos de datos.

---

13 Cuándo usar `__new__()` frente a `__init__()`

`__new__()` debe usarse siempre que desee personalizar el valor real del miembro `Enum`. Cualquier otra modificación puede ir en `__new__()` o `__init__()`, siendo preferible `__init__()`.

Por ejemplo, si desea pasar varios elementos al constructor, pero solo desea que uno de ellos sea el valor:

```
>>> 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
```

Advertencia: *No* llame a `super().__new__()`, ya que encontrará el `__new__` de solo búsqueda; en su lugar, utilice directamente el tipo de datos.

13.1 Puntos más finos

Nombres `__dunder__` admitidos

`__members__` es una asignación ordenada de solo lectura de elementos `member_name:member`. Solo está disponible en la clase.

`__new__()`, si se especifica, debe crear y devolver los miembros de enumeración; también es una muy buena idea configurar correctamente el `_value_` del miembro. Una vez que se crean todos los miembros, ya no se utiliza.

Nombres `_sunder_` admitidos

- `_name_` – nombre del miembro
- `_value_` – valor del miembro; se puede configurar/modificar en `__new__`
- `_missing_`: una función de búsqueda utilizada cuando no se encuentra un valor; puede ser anulado
- `_ignore_` – una lista de nombres, ya sea como `list` o `str`, que no se transformarán en miembros y se eliminarán de la clase final

- `_order_`: se usa en el código Python 2/3 para garantizar que el orden de los miembros sea coherente (atributo de clase, eliminado durante la creación de la clase)
- `_generate_next_value_`: utilizado por *Functional API* y por `auto` para obtener un valor apropiado para un miembro de enumeración; puede ser anulado

---

Nota: Para las clases `Enum` estándar, el siguiente valor elegido es el último valor visto incrementado en uno.

Para las clases `Flag`, el siguiente valor elegido será la siguiente potencia de dos más alta, independientemente del último valor visto.

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

Added in version 3.7: `_ignore_`

Para ayudar a mantener sincronizado el código de Python 2/Python 3, se puede proporcionar un atributo `_order_`. Se comparará con el orden real de la enumeración y lanzará un error si los dos no coinciden:

```
>>> 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: En el código de Python 2, el atributo `_order_` es necesario ya que el orden de definición se pierde antes de que se pueda registrar.

_Private__names

Private names no se convierten en miembros de enumeración, sino que siguen siendo atributos normales.

Distinto en la versión 3.11.

Tipo de miembro `Enum`

Los miembros de una enumeración son instancias de su clase de enumeración y se acceden normalmente como `EnumClass.member`. En ciertas situaciones, como al escribir comportamiento personalizado para una enumeración, es útil poder acceder a un miembro directamente desde otro, y esto está soportado; sin embargo, para evitar conflictos de nombres entre los nombres de los miembros y los atributos/métodos de las clases mezcladas, se recomienda encarecidamente utilizar nombres en mayúsculas.

Distinto en la versión 3.5.

Creación de miembros que se mezclan con otros tipos de datos

Al crear subclases de otros tipos de datos, como `int` o `str`, con un `Enum`, todos los valores después de = se pasan al constructor de ese tipo de datos. Por ejemplo:

```
>>> 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
```

Valor booleano de clases y miembros `Enum`

Las clases de enumeración que se mezclan con tipos que no son `Enum` (como `int`, `str`, etc.) se evalúan de acuerdo con las reglas del tipo combinado; de lo contrario, todos los miembros se evalúan como `True`. Para hacer que la evaluación booleana de su propia enumeración dependa del valor del miembro, agregue lo siguiente a su clase:

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

Las clases simples `Enum` siempre se evalúan como `True`.

Clases `Enum` con métodos

Si le da a su subclase de enumeración métodos adicionales, como la clase *Planet* a continuación, esos métodos aparecerán en un `dir()` del miembro, pero no de la clase:

```
>>> 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']
```

Combinación de miembros de `Flag`

La iteración sobre una combinación de miembros `Flag` solo devolverá los miembros que se componen de un solo 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>
```

Minuciosidades `Flag` y `IntFlag`

Usando el siguiente fragmento para nuestros ejemplos:

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

lo siguiente es cierto:

- las banderas de un solo bit son canónicas
- las banderas multibit y zero-bit son alias
- solo se retornan banderas canónicas durante la iteración:

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

- negar una bandera o un conjunto de banderas retorna una nueva bandera/conjunto de banderas con el valor entero positivo correspondiente:

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

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

- los nombres de las pseudo-banderas se construyen a partir de los nombres de sus miembros:

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

- las banderas de varios bits, también conocidas como alias, se pueden devolver desde las operaciones:

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

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

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

- comprobación de pertenencia / contención: las banderas de valor cero siempre se consideran contenidas:

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

  de lo contrario, solo si todos los bits de una bandera están en la otra bandera, se devolverá True:

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

(continúe en la próxima página)

(proviene de la página anterior)

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

Hay un nuevo mecanismo de límite que controla cómo se manejan los bits no válidos/fuera de rango: `STRICT`, `CONFORM`, `EJECT` y `KEEP`:

- STRICT –> lanza una excepción cuando se presentan valores no válidos
- CONFORM –> descarta cualquier bit inválido
- EJECT -> pierde el estado de la bandera y se convierte en un int normal con el valor dado
- KEEP –> mantener los bits adicionales
  - mantiene el estado de la bandera y bits adicionales
  - los bits adicionales no aparecen en la iteración
  - bits adicionales aparecen en repr() y str()

El valor predeterminado para Flag es `STRICT`, el valor predeterminado para `IntFlag` es `EJECT` y el valor predeterminado para `_convert_` es `KEEP` (consulte `ssl.Options` para ver un ejemplo de cuándo se necesita `KEEP`).

14 ¿En qué se diferencian las Enumeraciones (Enums) y las Banderas (Flags)?

Las enumeraciones tienen una metaclase personalizada que afecta a muchos aspectos de las clases `Enum` derivadas y sus instancias (miembros).

14.1 Clases de enumeración

La metaclase `EnumType` es responsable de proporcionar `__contains__()`, `__dir__()`, `__iter__()` y otros métodos que permiten hacer cosas con una clase `Enum` que fallan en una clase típica, como `list(Color)` o `some_enum_var in Color`. `EnumType` es responsable de garantizar que varios otros métodos en la clase `Enum` final sean correctos (como `__new__()`, `__getnewargs__()`, `__str__()` y `__repr__()`).

14.2 Clases de Banderas

Las banderas tienen una vista ampliada de la creación de alias: para ser canónico, el valor de una bandera debe ser un valor de potencia de dos y no un nombre duplicado. Por lo tanto, además de la definición de alias de `Enum`, una bandera sin valor (también conocida como `0`) o con más de un valor de potencia de dos (por ejemplo, `3`) se considera un alias.

14.3 Miembros de enumeración (también conocidos como instancias)

Lo más interesante de los miembros de la enumeración es que son únicos. `EnumType` los crea a todos mientras crea la propia clase de enumeración, y luego coloca un `__new__()` personalizado para garantizar que nunca se creen instancias nuevas al devolver solo las instancias de miembros existentes.

14.4 Miembros de Banderas

Los miembros de las Banderas se pueden recorrer de la misma manera que la clase `Flag`, y solo se devolverán los miembros canónicos. Por ejemplo:

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

(Note que `BLACK`, `PURPLE`, y `WHITE` no se muestran.)

Invertir un miembro de la bandera devuelve el valor positivo correspondiente, en lugar de un valor negativo — por ejemplo:

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

Los miembros de las Banderas tienen una longitud que corresponde al número de valores de potencia de dos que contienen. Por ejemplo:

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

15 Recetario de Enumeraciones

Si bien se espera que `Enum`, `IntEnum`, `StrEnum`, `Flag` y `IntFlag` cubran la mayoría de los casos de uso, no pueden cubrirlos todos. Aquí hay recetas para algunos tipos diferentes de enumeraciones que se pueden usar directamente o como ejemplos para crear las propias.

15.1 Omitir valores

En muchos casos de uso, a uno no le importa cuál es el valor real de una enumeración. Hay varias formas de definir este tipo de enumeración simple:

- usar instancias de `auto` para el valor
- usar instancias de `object` como valor
- use una cadena descriptiva como el valor
- use una tupla como valor y un `__new__()` personalizado para reemplazar la tupla con un valor `int`

El uso de cualquiera de estos métodos significa para el usuario que estos valores no son importantes y también permite agregar, eliminar o reordenar miembros sin tener que volver a numerar los miembros restantes.

Usando `auto`

El uso de `auto` se vería así:

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

Usando `object`

El uso de `object` se vería así:

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

Este también es un buen ejemplo de por qué es posible que desee escribir su propio `__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>
```

Usar una cadena descriptiva

Usando una cadena como el valor se vería así:

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

Usando un `__new__()` personalizado

El uso de un `__new__()` de numeración automática se vería así:

```
>>> 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>
```

Para hacer un `AutoNumber` de uso más general, agregue `*args` a la firma:

```
>>> 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
...
```

Luego, cuando hereda de `AutoNumber`, puede escribir su propio `__init__` para manejar cualquier argumento adicional:

```
>>> 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: El método `__new__()`, si está definido, se usa durante la creación de los miembros de Enum; luego se reemplaza por `__new__()` de Enum, que se usa después de la creación de clases para buscar miembros existentes.

Advertencia: *No* llame a `super().__new__()`, ya que encontrará el `__new__` de solo búsqueda; en su lugar, utilice directamente el tipo de datos – por ejemplo:

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

15.2 Enum ordenado

Una enumeración ordenada que no se basa en `IntEnum` y, por lo tanto, mantiene las invariantes normales de `Enum` (como no ser comparable con otras enumeraciones):

```
>>> 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

Lanza un error si se encuentra un nombre de miembro duplicado en lugar de crear un 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
...
Traceback (most recent call last):
  ...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE' --> 'GREEN'
```

Nota: Este es un ejemplo útil para subclasificar Enum para agregar o cambiar otros comportamientos, así como para no permitir alias. Si el único cambio deseado es prohibir los alias, se puede usar el decorador `unique()` en su lugar.

15.4 Planeta

Si se define `__new__()` o `__init__()`, el valor del miembro de enumeración se pasará a esos métodos:

```
>>> 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 Periodo de tiempo

Un ejemplo para mostrar el atributo `_ignore_` en uso:

```
>>> 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 Subclase EnumType

Si bien la mayoría de las necesidades de enumeración se pueden satisfacer mediante la personalización de las subclases `Enum`, ya sea con decoradores de clase o funciones personalizadas, `EnumType` se puede dividir en subclases para proporcionar una experiencia de enumeración diferente.