enum— Suporte para enumerações

Adicionado na versão 3.4.

Código fonte: Lib/enum.py

Uma enumeração:

- é um conjunto de nomes simbólicos (membros) vinculados a valores únicos
- pode ser iterado para retornar seus membros canônicos (ou seja, não alias) na ordem de definição
- usa sintaxe *de chamada* para retornar membros por valor
- usa sintaxe *de índice* para retornar membros por nome

As enumerações são criadas usando `class` sintaxe ou usando sintaxe de chamada de função:

Importante

Esta página contém as informações de referência da API. Para obter informações do tutorial e discussão de tópicos mais avançados, consulte

- Tutorial Básico
- Tutorial avançado
- Livro de receitas Enum

```
>>> from enum import Enum

>>> # class syntax
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3

>>> # functional syntax
>>> Color = Enum('Color', [('RED', 1), ('GREEN', 2), ('BLUE', 3)])
```

Embora possamos usar `class` sintaxe para criar Enums, Enums não são classes Python normais. Veja Como os Enums são diferentes? para mais detalhes.

Observação: Nomenclatura

- A classe `Color` é uma *enumeração* (ou *enum*)
- Os atributos `Color.RED`, `Color.GREEN`, etc., são *membros de enumeração* (ou *membros*) e são funcionalmente constantes.
- Os membros enum têm *nomes* e *valores* (o nome de `Color.RED` is `RED`, o valor de `Color.BLUE` is `3`, etc.)

Conteúdo do módulo

`EnumType`

O `type` for Enum e suas subclasses.

`Enum`

`IntEnum`

Classe base para criar constantes enumeradas que também são subclasses de `int`. (Notas)

`StrEnum`

Classe base para criar constantes enumeradas que também são subclasses de `str`. (Notas)

`Flag`

Classe base para criar constantes enumeradas que podem ser combinadas usando operações bit a bit sem perder sua `Flag` associação.

`IntFlag`

Classe base para criar constantes enumeradas que podem ser combinadas usando operadores bit a bit sem perder sua `IntFlag` associação. `IntFlag` Os membros também são subclasses de `int`. (Notas)

`ReprEnum`

Usado por `IntEnum`, `StrEnum`, e `IntFlag` para manter o `str()` do tipo misto.

`EnumCheck`

Uma enumeração com os valores `CONTINUOUS`, `NAMED_FLAGS`, e `UNIQUE`, para uso com `verify()` para garantir que várias restrições sejam atendidas por uma determinada enumeração.

`FlagBoundary`

An enumeration with the values `STRICT`, `CONFORM`, `EJECT`, and `KEEP` which allows for more fine-grained control over how invalid values are dealt with in an enumeration.

`auto`

Instances are replaced with an appropriate value for Enum members. `StrEnum` defaults to the lower-cased version of the member name, while other Enums default to 1 and increase from there.

`property()`

Allows `Enum` members to have attributes without conflicting with member names. The `value` and `name` attributes are implemented this way.

`unique()`

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

`verify()`

Enum class decorator that checks user-selectable constraints on an enumeration.

`member()`

`nonmember()`

Do not make `obj` a member. Can be used as a decorator.

`global_enum()`

Modify the `str()` and `repr()` of an enum to show its members as belonging to the module instead of its class, and export the enum members to the global namespace.

`show_flag_values()`

Return a list of all power-of-two integers contained in a flag.

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

Added in version 3.11: `StrEnum`, `EnumCheck`, `ReprEnum`, `FlagBoundary`, `property`, `member`, `nonmember`, `global_enum`, `show_flag_values`

Data Types

class enum.**EnumType**

EnumType is the metaclass for *enum* enumerations. It is possible to subclass *EnumType* – see Subclassing EnumType for details.

`EnumType` is responsible for setting the correct `__repr__()`, `__str__()`, `__format__()`, and `__reduce__()` methods on the final *enum*, as well as creating the enum members, properly handling duplicates, providing iteration over the enum class, etc.

__call__(*cls, value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None*)

This method is called in two different ways:

- to look up an existing member:

  cls: The enum class being called.
  value: The value to lookup.

- to use the `cls` enum to create a new enum (only if the existing enum does not have any members):

  cls: The enum class being called.
  value: The name of the new Enum to create.
  names: The names/values of the members for the new Enum.
  module: The name of the module the new Enum is created in.
  qualname: The actual location in the module where this Enum can be found.
  type: A mix-in type for the new Enum.
  start: The first integer value for the Enum (used by `auto`).
  boundary: How to handle out-of-range values from bit operations (`Flag` only).

Returns True if member belongs to the `cls`:

```
>>> some_var = Color.RED
>>> some_var in Color
True
>>> Color.RED.value in Color
True
```

Changed in version 3.12: Before Python 3.12, a `TypeError` is raised if a non-Enum-member is used in a containment check.

`__dir__`(*cls*)

Returns `['__class__', '__doc__', '__members__', '__module__']` and the names of the members in *cls*:

```
>>> dir(Color)
['BLUE', 'GREEN', 'RED', '__class__', '__contains__', '__doc__', '__getitem__
```

`__getitem__`(*cls*, *name*)

Returns the Enum member in *cls* matching *name*, or raises a `KeyError`:

```
>>> Color['BLUE']
<Color.BLUE: 3>
```

`__iter__`(*cls*)

Returns each member in *cls* in definition order:

```
>>> list(Color)
[<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 3>]
```

`__len__`(*cls*)

Returns the number of member in *cls*:

```
>>> len(Color)
3
```

`__members__`

Returns a mapping of every enum name to its member, including aliases

`__reversed__`(*cls*)

Returns each member in *cls* in reverse definition order:

```
>>> list(reversed(Color))
[<Color.BLUE: 3>, <Color.GREEN: 2>, <Color.RED: 1>]
```

`_add_alias_`()

Adds a new name as an alias to an existing member. Raises a `NameError` if the name is already assigned to a different member.

Adds a new value as an alias to an existing member. Raises a `ValueError` if the value is already linked with a different member.

> *Added in version 3.11:* Before 3.11 `EnumType` was called `EnumMeta`, which is still available as an alias.

class enum.**Enum**

Enum is the base class for all *enum* enumerations.

name

The name used to define the `Enum` member:

```
>>> Color.BLUE.name
'BLUE'
```

value

The value given to the `Enum` member:

```
>>> Color.RED.value
1
```

Value of the member, can be set in `__new__()`.

> **Note:** Enum member values
>
> Member values can be anything: `int`, `str`, etc. If the exact value is unimportant you may use `auto` instances and an appropriate value will be chosen for you. See `auto` for the details.
>
> While mutable/unhashable values, such as `dict`, `list` or a mutable `dataclass`, can be used, they will have a quadratic performance impact during creation relative to the total number of mutable/unhashable values in the enum.

name

Name of the member.

value

Value of the member, can be set in `__new__()`.

order

No longer used, kept for backward compatibility. (class attribute, removed during class creation).

ignore

`_ignore_` is only used during creation and is removed from the enumeration once creation is complete.

`_ignore_` is a list of names that will not become members, and whose names will also be removed from the completed enumeration. See TimePeriod for an example.

Returns [`__class__`, `__doc__`, `__module__`, `name`, `value`] and any public methods defined on *self.__class__*:

```
>>> from datetime import date
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...     @classmethod
...     def today(cls):
...         print('today is %s' % cls(date.today().isoweekday()).name)
...
>>> dir(Weekday.SATURDAY)
['__class__', '__doc__', '__eq__', '__hash__', '__module__', 'name', 'today',
```

`_generate_next_value_(name, start, count, last_values)`

name:	The name of the member being defined (e.g. 'RED').
start:	The start value for the Enum; the default is 1.
count:	The number of members currently defined, not including this one.
last_values:	A list of the previous values.

A *staticmethod* that is used to determine the next value returned by `auto`:

```
>>> from enum import auto
>>> class PowersOfThree(Enum):
...     @staticmethod
...     def _generate_next_value_(name, start, count, last_values):
...         return 3 ** (count + 1)
...     FIRST = auto()
...     SECOND = auto()
...
>>> PowersOfThree.SECOND.value
9
```

`__init__(self, *args, **kwds)`

By default, does nothing. If multiple values are given in the member assignment, those values become separate arguments to `__init__`; e.g.

```
>>> from enum import Enum
>>> class Weekday(Enum):
...     MONDAY = 1, 'Mon'
```

`Weekday.__init__()` would be called as `Weekday.__init__(self, 1, 'Mon')`

`__init_subclass__(cls, **kwds)`

A *classmethod* that is used to further configure subsequent subclasses. By default, does nothing.

`_missing_(cls, value)`

```
>>> from enum import StrEnum
>>> class Build(StrEnum):
...     DEBUG = auto()
...     OPTIMIZED = auto()
...     @classmethod
...     def _missing_(cls, value):
...         value = value.lower()
...         for member in cls:
...             if member.value == value:
...                 return member
...         return None
...
>>> Build.DEBUG.value
'debug'
>>> Build('deBUG')
<Build.DEBUG: 'debug'>
```

`__new__(cls, *args, **kwds)`

By default, doesn't exist. If specified, either in the enum class definition or in a mixin class (such as `int`), all values given in the member assignment will be passed; e.g.

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     TWENTYSIX = '1a', 16
```

results in the call `int('1a', 16)` and a value of `26` for the member.

Note: When writing a custom `__new__`, do not use `super().__new__` – call the appropriate `__new__` instead.

`__repr__(self)`

Returns the string used for *repr()* calls. By default, returns the *Enum* name, member name, and value, but can be overridden:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __repr__(self):
...         cls_name = self.__class__.__name__
...         return f'{cls_name}.{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}
(OtherStyle.ALTERNATE, 'OtherStyle.ALTERNATE', 'OtherStyle.ALTERNATE')
```

`__str__(self)`

Returns the string used for *str()* calls. By default, returns the *Enum* name and member name, but can be overridden:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
```

```
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}
(<OtherStyle.ALTERNATE: 1>, 'ALTERNATE', 'ALTERNATE')
```

__format__(self)

Returns the string used for *format()* and *f-string* calls. By default, returns `__str__()` return value, but can be overridden:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __format__(self, spec):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}
(<OtherStyle.ALTERNATE: 1>, 'OtherStyle.ALTERNATE', 'ALTERNATE')
```

Note: Using `auto` with `Enum` results in integers of increasing value, starting with `1`.

Changed in version 3.12: Added Dataclass support

class enum.**IntEnum**

IntEnum is the same as `Enum`, but its members are also integers and can be used anywhere that an integer can be used. If any integer operation is performed with an *IntEnum* member, the resulting value loses its enumeration status.

```
>>> from enum import IntEnum
>>> class Number(IntEnum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...
>>> Number.THREE
<Number.THREE: 3>
>>> Number.ONE + Number.TWO
3
>>> Number.THREE + 5
8
>>> Number.THREE == 3
True
```

Note: Using `auto` with `IntEnum` results in integers of increasing value, starting with `1`.

Changed in version 3.11: `__str__()` is now `int.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` was already `int.__format__()` for that same reason.

class enum.**StrEnum**

ber is not part of the enumeration.

Note: There are places in the stdlib that check for an exact `str` instead of a `str` subclass (i.e. `type(unknown) == str` instead of `isinstance(unknown, str)`), and in those locations you will need to use `str(StrEnum.member)`.

Note: Using `auto` with `StrEnum` results in the lower-cased member name as the value.

Note: `__str__()` is `str.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` is likewise `str.__format__()` for that same reason.

Added in version 3.11.

class enum.**Flag**

`Flag` is the same as `Enum`, but its members support the bitwise operators `&` (*AND*), `|` (*OR*), `^` (*XOR*), and `~` (*INVERT*); the results of those operations are (aliases of) members of the enumeration.

__contains__(*self*, *value*)

Returns *True* if value is in self:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> purple = Color.RED | Color.BLUE
>>> white = Color.RED | Color.GREEN | Color.BLUE
>>> Color.GREEN in purple
False
>>> Color.GREEN in white
True
>>> purple in white
True
>>> white in purple
False
```

__iter__(self):

Returns all contained non-alias members:

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

Added in version 3.11.

__len__(self):

Returns number of members in flag:

```
>>> len(white)
3
```

Added in version 3.11.

`__bool__(self):`

Returns *True* if any members in flag, *False* otherwise:

```
>>> bool(Color.GREEN)
True
>>> bool(white)
True
>>> black = Color(0)
>>> bool(black)
False
```

`__or__(self, other)`

Returns current flag binary or'ed with other:

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

`__and__(self, other)`

Returns current flag binary and'ed with other:

```
>>> purple & white
<Color.RED|BLUE: 5>
>>> purple & Color.GREEN
<Color: 0>
```

`__xor__(self, other)`

Returns current flag binary xor'ed with other:

```
>>> purple ^ white
<Color.GREEN: 2>
>>> purple ^ Color.GREEN
<Color.RED|GREEN|BLUE: 7>
```

`__invert__(self):`

Returns all the flags in *type(self)* that are not in *self*:

```
>>> ~white
<Color: 0>
>>> ~purple
<Color.GREEN: 2>
>>> ~Color.RED
<Color.GREEN|BLUE: 6>
```

`_numeric_repr_()`

Function used to format any remaining unnamed numeric values. Default is the value's repr; common choices are `hex()` and `oct()`.

> *Changed in version 3.11:* The *repr()* of zero-valued flags has changed. It is now::
>
> ```
> >>> Color(0)
> <Color: 0>
> ```

class enum.**IntFlag**

`IntFlag` is the same as `Flag`, but its members are also integers and can be used anywhere that an integer can be used.

```
>>> from enum import IntFlag, auto
>>> class Color(IntFlag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> Color.RED & 2
<Color: 0>
>>> Color.RED | 2
<Color.RED|GREEN: 3>
```

If any integer operation is performed with an *IntFlag* member, the result is not an *IntFlag*:

```
>>> Color.RED + 2
3
```

If a `Flag` operation is performed with an *IntFlag* member and:

- the result is a valid *IntFlag*: an *IntFlag* is returned
- the result is not a valid *IntFlag*: the result depends on the `FlagBoundary` setting

The `repr()` of unnamed zero-valued flags has changed. It is now:

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

Note: Using `auto` with `IntFlag` results in integers that are powers of two, starting with `1`.

> *Changed in version 3.11:* `__str__()` is now `int.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` was already `int.__format__()` for that same reason.
>
> Inversion of an `IntFlag` now returns a positive value that is the union of all flags not in the given flag, rather than a negative value. This matches the existing `Flag` behavior.

class enum.**ReprEnum**

`ReprEnum` uses the `repr()` of `Enum`, but the `str()` of the mixed-in data type:

- `int.__str__()` for `IntEnum` and `IntFlag`
- `str.__str__()` for `StrEnum`

Added in version 3.11.

class enum.**EnumCheck**

EnumCheck contains the options used by the `verify()` decorator to ensure various constraints; failed constraints result in a `ValueError`.

UNIQUE

Ensure that each value has only one name:

```
>>> from enum import Enum, verify, UNIQUE
>>> @verify(UNIQUE)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     CRIMSON = 1
Traceback (most recent call last):
...
ValueError: aliases found in <enum 'Color'>: CRIMSON -> RED
```

CONTINUOUS

Ensure that there are no missing values between the lowest-valued member and the highest-valued member:

```
>>> from enum import Enum, verify, CONTINUOUS
>>> @verify(CONTINUOUS)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 5
Traceback (most recent call last):
...
ValueError: invalid enum 'Color': missing values 3, 4
```

NAMED_FLAGS

Ensure that any flag groups/masks contain only named flags – useful when values are specified instead of being generated by `auto()`:

```
>>> from enum import Flag, verify, NAMED_FLAGS
>>> @verify(NAMED_FLAGS)
... class Color(Flag):
...     RED = 1
...     GREEN = 2
...     BLUE = 4
...     WHITE = 15
...     NEON = 31
Traceback (most recent call last):
...
ValueError: invalid Flag 'Color': aliases WHITE and NEON are missing combined
```

Note: CONTINUOUS and NAMED_FLAGS are designed to work with integer-valued members.

class enum.**FlagBoundary**

`FlagBoundary` controls how out-of-range values are handled in `Flag` and its subclasses.

STRICT

Out-of-range values cause a `ValueError` to be raised. This is the default for `Flag`:

```
>>> from enum import Flag, STRICT, auto
>>> class StrictFlag(Flag, boundary=STRICT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> StrictFlag(2**2 + 2**4)
Traceback (most recent call last):
...
ValueError: <flag 'StrictFlag'> invalid value 20
    given 0b0 10100
  allowed 0b0 00111
```

CONFORM

Out-of-range values have invalid values removed, leaving a valid `Flag` value:

```
>>> from enum import Flag, CONFORM, auto
>>> class ConformFlag(Flag, boundary=CONFORM):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> ConformFlag(2**2 + 2**4)
<ConformFlag.BLUE: 4>
```

EJECT

Out-of-range values lose their `Flag` membership and revert to `int`.

```
>>> from enum import Flag, EJECT, auto
>>> class EjectFlag(Flag, boundary=EJECT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> EjectFlag(2**2 + 2**4)
20
```

KEEP

Out-of-range values are kept, and the `Flag` membership is kept. This is the default for `IntFlag`:

```
>>> from enum import Flag, KEEP, auto
>>> class KeepFlag(Flag, boundary=KEEP):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> KeepFlag(2**2 + 2**4)
<KeepFlag.BLUE|16: 20>
```

Supported `__dunder__` names

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

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

Supported `_sunder_` names

- `_add_alias_()` – adds a new name as an alias to an existing member.
- `_add_value_alias_()` – adds a new value as an alias to an existing member.
- `_name_` – name of the member
- `_value_` – value of the member; can be set in `__new__`
- `_missing_()` – a lookup function used when a value is not found; may be overridden
- `_ignore_` – a list of names, either as a `list` or a `str`, that will not be transformed into members, and will be removed from the final class
- `_order_` – no longer used, kept for backward compatibility (class attribute, removed during class creation)
- `_generate_next_value_()` – used to get an appropriate value for an enum member; may be overridden

  Note: For standard `Enum` classes the next value chosen is the highest value seen incremented by one.

  For `Flag` classes the next value chosen will be the next highest power-of-two.

- While `_sunder_` names are generally reserved for the further development of the `Enum` class and can not be used, some are explicitly allowed:
  - `_repr_*` (e.g. `_repr_html_`), as used in IPython's rich display

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_`, `_repr_*`

Utilities and Decorators

class enum.**auto**

auto can be used in place of a value. If used, the *Enum* machinery will call an `Enum`'s `_generate_next_value_()` to get an appropriate value. For `Enum` and `IntEnum` that appropriate value will be the last value plus one; for `Flag` and `IntFlag` it will be the first power-of-two greater than the

auto instances are only resolved when at the top level of an assignment:

- `FIRST = auto()` will work (auto() is replaced with `1`);
- `SECOND = auto(), -2` will work (auto is replaced with `2`, so `2, -2` is used to create the `SECOND` enum member;
- `THREE = [auto(), -3]` will *not* work (`<auto instance>, -3` is used to create the `THREE` enum member)

> *Changed in version 3.11.1:* In prior versions, `auto()` had to be the only thing on the assignment line to work properly.

`_generate_next_value_` can be overridden to customize the values used by *auto*.

> **Note:** in 3.13 the default `_generate_next_value_` will always return the highest member value incremented by 1, and will fail if any member is an incompatible type.

`@enum.property`

A decorator similar to the built-in *property*, but specifically for enumerations. It allows member attributes to have the same names as members themselves.

> **Note:** the *property* and the member must be defined in separate classes; for example, the *value* and *name* attributes are defined in the *Enum* class, and *Enum* subclasses can define members with the names `value` and `name`.

> *Added in version 3.11.*

`@enum.unique`

A `class` decorator specifically for enumerations. It searches an enumeration's `__members__`, gathering any aliases it finds; if any are found `ValueError` is raised with the details:

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

`@enum.verify`

A `class` decorator specifically for enumerations. Members from `EnumCheck` are used to specify which constraints should be checked on the decorated enumeration.

> *Added in version 3.11.*

`@enum.member`

Added in version 3.11.

`@enum.`**`nonmember`**

A decorator for use in enums: its target will not become a member.

Added in version 3.11.

`@enum.`**`global_enum`**

A decorator to change the `str()` and `repr()` of an enum to show its members as belonging to the module instead of its class. Should only be used when the enum members are exported to the module global namespace (see `re.RegexFlag` for an example).

Added in version 3.11.

`enum.show_flag_values` `(` **`valor`** `)`

Retorna uma lista de todos os inteiros de potência de dois contidos em um *valor* de sinalizador .

Adicionado na versão 3.11.

Notas

`IntEnum`, `StrEnum`, e`IntFlag`

Esses três tipos de enumeração foram projetados para serem substituições imediatas para valores existentes baseados em inteiros e strings; como tal, eles têm limitações extras:

- `__str__`usa o valor e não o nome do membro enum
- `__format__`, porque ele usa `__str__`, também usará o valor do membro enum em vez de seu nome

Se você não precisa/não quer essas limitações, você pode criar sua própria classe base misturando `int`ou `str`digitando você mesmo:

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     pass
```

ou você pode reatribuir o apropriado `str()`, etc., em seu enum:

```
>>> from enum import Enum, IntEnum
>>> class MyIntEnum(IntEnum):
...     __str__ = Enum.__str__
```