

---

# 列挙型 HOWTO

リリース 3.11.14

Guido van Rossum and the Python development team

10 月 15, 2025

## 目次

1	列挙型メンバーおよびそれらの属性へのプログラムのアクセス	6
2	列挙型メンバーと値の重複	6
3	番号付けの値が一意であることの確認	7
4	値の自動設定を使う	7
5	イテレーション	8
6	比較	9
7	列挙型で許されるメンバーと属性	10
8	Enum のサブクラス化の制限	11
9	Pickle 化	12
10	関数 API	12
11	派生列挙型	14
11.1	IntEnum . . . . .	14
11.2	StrEnum . . . . .	15
11.3	IntFlag . . . . .	15
11.4	Flag . . . . .	17
11.5	その他 . . . . .	19
12	When to use <code>__new__()</code> vs. <code>__init__()</code>	19
12.1	細かい点 . . . . .	20

13	Enum と Flag はどう違うのか？	25
13.1	Enum クラス	25
13.2	Flag クラス	25
13.3	Enum メンバー (インスタンス)	25
13.4	Flag メンバー	25
14	Enum クックブック	26
14.1	値の省略	26
14.2	OrderedEnum	29
14.3	DuplicateFreeEnum	29
14.4	Planet	30
14.5	TimePeriod	31
15	EnumType のサブクラスを作る	31

---

An **Enum** is a set of symbolic names bound to unique values. They are similar to global variables, but they offer a more useful `repr()`, grouping, type-safety, and a few other features.

これらは、限られた選択枝の値の一つを取る変数がある場合に便利です。例えば、曜日情報があります:

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

あるいは、RGB 三原色でも構いません:

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

ご覧の通り、Enum の作成は Enum 自体を継承するクラスを作成するのと同じくらい簡単です。

---

**注釈:** Enum メンバーは大文字/小文字？

Because Enums are used to represent constants we recommend using UPPER\_CASE names for members,

and will be using that style in our examples.

---

列挙型の性質によって、メンバの値は重要な場合とそうでない場合がありますが、いずれの場合でも、その値は対応するメンバを取得するのに使えます:

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

ご覧の通り、メンバの `repr()` は列挙型の名前、メンバの名前、そして値を表示します。メンバの `str()` は列挙型の名前とメンバの名前のみを表示します。

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

列挙型のメンバの型はそのメンバの属する列挙型です:

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

Enum members have an attribute that contains just their **name**:

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

Likewise, they have an attribute for their **value**:

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

Unlike many languages that treat enumerations solely as name/value pairs, Python Enums can have behavior added. For example, `datetime.date` has two methods for returning the weekday: `weekday()` and `isoweekday()`. The difference is that one of them counts from 0-6 and the other from 1-7. Rather than keep track of that ourselves we can add a method to the `Weekday` enum to extract the day from the `date` instance and return the matching enum member:

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

The complete `Weekday` enum now looks like this:

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

さて、これで今日が何曜日か調べることができます！ 見てみましょう：

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

もちろん、あなたがこれを読んでいるのが他の曜日ならば、その曜日が代わりに表示されます。

This Weekday enum is great if our variable only needs one day, but what if we need several? Maybe we're writing a function to plot chores during a week, and don't want to use a `list` -- we could use a different type of Enum:

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

ここでは2つの変更が行われています。Flag を継承している点と、値がすべて2の累乗である点です。

Just like the original Weekday enum above, we can have a single selection:

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

ただし、Flag は複数のメンバーをひとつの変数にまとめることもできます：

```
>>> weekend = Weekday.SATURDAY | Weekday.SUNDAY
>>> weekend
```

(次のページに続く)

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

Flag 変数は反復することもできます:

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

さて、いくつかの家事を設定してみましょう:

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

指定された日の家事を表示する関数も作成します:

```
>>> 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
```

In cases where the actual values of the members do not matter, you can save yourself some work and use `auto()` for the values:

```
>>> 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 列挙型メンバーおよびそれらの属性へのプログラムのアクセス

プログラムのメンバーに番号でアクセスしたほうが便利な場合があります (すなわち、プログラムを書いている時点で正確な色がまだわからなく、`Color.RED` と書くのが無理な場合など)。Enum ではそのようなアクセスも可能です:

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

列挙型メンバーに **名前** でアクセスしたい場合はアイテムとしてアクセスできます:

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

If you have an enum member and need its **name** or **value**:

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

## 2 列挙型メンバーと値の重複

同じ名前の列挙型メンバーを複数持つことはできません:

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

しかし、列挙型メンバーは、別の名前を持つことができます。同じ値を持つ A と “B” が与えられた場合 (そして A が先に定義されている場合)、B はメンバー A に対するエイリアスとなります。A の値での検索では、メンバー A が返されます。A の名前での検索ではメンバー A を返します。B の名前での検索も、メンバー A を返します:

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

(次のページに続く)

(前のページからの続き)

```
...     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>
```

---

**注釈:** すでに定義されている属性と同じ名前のメンバー (一方がメンバーでもう一方がメソッド、など) の作成、あるいはメンバーと同じ名前の属性の作成はできません。

---

### 3 番号付けの値が一意であることの確認

デフォルトでは、列挙型は同じ値のエイリアスとして複数の名前を許容します。この振る舞いを望まない場合は、`unique()` デコレータを使用できます:

```
>>> 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 値の自動設定を使う

正確な値が重要でない場合、`auto` が使えます:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
```

(次のページに続く)

(前のページからの続き)

```
...
>>> [member.value for member in Color]
[1, 2, 3]
```

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

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

---

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

---

## 5 イテレーション

列挙型のメンバーのイテレートは別名をサポートしていません:

```
>>> 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>]
```

エイリアスである `Shape.ALIAS_FOR_SQUARE` と “`Weekday.WEEKEND`” が表示されていないことに注意してください。

特殊属性 `__members__` は読み出し専用で、順序を保持した、対応する名前と列挙型メンバーのマッピングです。これには別名も含め、列挙されたすべての名前が入っています。

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
```

(次のページに続く)



(前のページからの続き)

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

属性 `__members__` は列挙型メンバーへの詳細なアクセスに使用できます。以下はすべての別名を探す例です:

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

---

**注釈:** フラグのエイリアスには、複数のフラグが設定された値（例えば 3）や、フラグが設定されていない値（例えば 0）が含まれます。

---

## 6 比較

列挙型メンバーは同一性を比較できます:

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

列挙型の値の順序の比較はサポートされて **いません**。Enum メンバーは整数ではありません (*IntEnum* を参照してください):

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

ただし等価の比較は定義されています:

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

非列挙型の値との比較は常に不等となります (繰り返しになりますが、*IntEnum* はこれと異なる挙動になるよう設計されています):

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

**警告:** モジュールは再読み込みすることが可能です。再読み込みされたモジュールに列挙型が含まれている場合、それらは再作成され、新しいメンバーは元のメンバーと同一でない/等しくない可能性があります。

## 7 列挙型で許されるメンバーと属性

これまでのほとんどの例では、列挙型の値に整数を使用しています。整数を使うのは短くて便利（そして、[関数 API](#) ではデフォルトで設定される）ですが、これは強制されているわけではありません。大半の使用例では、列挙値の実際の値が何であるかは意識しません。しかし、値が重要な場合、列挙型は任意の値を持つことができます。

列挙型は Python のクラスであり、通常どおりメソッドや特殊メソッドを持つことができます:

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

上記の結果が以下ようになります:

```
>>> 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.

**注釈:** 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 Enum のサブクラス化の制限

新しい Enum クラスは、ベースの enum クラスを1つ、具象データ型を1つ、複数の object ベースのミックスインクラスが許容されます。これらのベースクラスの順序は次の通りです:

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

列挙型のサブクラスの作成はその列挙型にメンバーが一つも定義されていない場合のみ行なえます。従って以下は許されません:

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

以下のような場合は許されます:

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

メンバーが定義された列挙型のサブクラス化を許可すると、いくつかのデータ型およびインスタンスの重要な不変条件の違反を引き起こします。とはいえ、それが許可されると、列挙型のグループ間での共通の挙動を共有するという利点もあります。( *OrderedEnum* の例を参照してください。 )

## 9 Pickle 化

列挙型は pickle 化と unpickle 化が行えます:

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

通常の pickle 化の制限事項が適用されます: pickle 可能な列挙型はモジュールのトップレベルで定義されていなくてはならず、unpickle 化はモジュールからインポート可能でなければなりません。

---

**注釈:** pickle プロトコルバージョン 4 では他のクラスで入れ子になった列挙型の pickle 化も容易です。

---

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
```

---

**注釈:** フラグに名前による pickle 化を使うことは、名前の無いエイリアスが unpickle 化されないため推奨されません。

---

## 10 関数 API

Enum クラスは呼び出し可能で、以下の関数 API を提供しています:

```
>>> 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>]
```

この API の動作は `namedtuple` と似ています。Enum 呼び出しの第 1 引数は列挙型の名前です。

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
... 
```

0 ではなく “1” をデフォルトの開始番号とする理由は、0 が真偽値としては `False` であり、デフォルトの列挙メンバーはすべて `True` 評価されるようにするためである。

機能 API による `Enum` の pickle 化は、その列挙型がどのモジュールで作成されたかを見つけ出すためにフレームスタックの実装の詳細が使われるので、トリッキーになることがあります (例えば別のモジュールのユーティリティ関数を使うと失敗しますし、IronPython や Jython ではうまくいきません)。解決策は、以下のようにモジュール名を明示的に指定することです:

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

**警告:** `module` が与えられない場合、`Enum` はそれがなにか決定できないため、新しい `Enum` メンバーは unpickle 化できなくなります; エラーをソースの近いところで発生させるため、pickle 化は無効になります。

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')
```

完全な構文は以下のようになります:

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

- *value*: What the new enum class will record as its name.

- *names*: enum のメンバー。空白またはカンマで区切られた文字列 (値は特に指定がない限り 1 から始まります):

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

または名前のイテレータで指定もできます:

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

または (名前, 値) のペアのイテレータでも指定できます:

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

またはマッピングでも指定できます:

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

- *module*: 新しい enum クラスが属するモジュールの名前です。
- *qualname*: 新しい enum クラスが属するモジュールの場所です。
- *type*: 新しい列挙型クラスにミックスインする型。
- *start*: 名前だけ渡された場合にカウントを開始する番号。

バージョン 3.5 で変更: *start* 引数が追加されました。

## 11 派生列挙型

### 11.1 IntEnum

提供されている 1 つ目の Enum の派生型であり、int のサブクラスでもあります。IntEnum のメンバーは整数と比較できます; さらに言うと、異なる整数列挙型どうしても比較できます:

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

(次のページに続く)

(前のページからの続き)

```
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

ただし、これらも標準の Enum 列挙型とは比較できません:

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

IntEnum の値は他の用途では整数のように振る舞います:

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

## 11.2 StrEnum

提供されている 2 つ目の Enum の派生型もまた、str のサブクラスでもあります。StrEnum のメンバーは、文字列と比較できます; さらに言うと、異なる文字列列挙型どうしても比較できます。

バージョン 3.11 で追加.

## 11.3 IntFlag

次の Enum の派生型 IntFlag も、int を基底クラスとしています。違いは、IntFlag のメンバーをビット演算子 (&, |, ^, ~) を使って組み合わせられ、その結果も IntFlag メンバーになることです。IntEnum と同様、IntFlag のメンバーも整数であり、int が使用される場所であればどこでも使えます。

---

**注釈:** IntFlag メンバーに対してビット演算以外のどんな演算をしても、その結果は IntFlag メンバーではありません。

ビット単位演算の結果が `IntFlag` として不正な値の場合、値は `IntFlag` メンバーではなくなります。詳しくは `FlagBoundary` を参照してください。

---

バージョン 3.6 で追加.

バージョン 3.11 で変更.

`IntFlag` クラスの例:

```
>>> 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
```

組み合わせにも名前を付けられます:

```
>>> 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>
```

---

**注釈:** 組み合わせに名前をつけたものはエイリアスとみなされます。エイリアスはイテレーション中には表示されませんが、値による検索では返却されます。

---

バージョン 3.11 で変更.

`IntFlag` と `Enum` のもう 1 つの重要な違いは、フラグが設定されていない (値が 0 である) 場合、その真偽値としての評価は `False` になることです:



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

IntFlag メンバーも int のサブクラスであるため、それらと組み合わせることができます（ただし、IntFlag 型ではなくなる可能性があります）：

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

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

---

**注釈：** 否定の演算子 ~ は、常に正の値を持つ IntFlag メンバー を返す：

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

---

IntFlag メンバーは反復処理することもできます：

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

バージョン 3.11 で追加.

## 11.4 Flag

最後の派生型は Flag です。IntFlag と同様に、Flag メンバーもビット演算子 (&, |, ^, ~) を使って組み合わせられます。しかし IntFlag とは違い、他のどの Flag 列挙型とも int と組み合わせたり、比較したりできません。値を直接指定することも可能ですが、値として auto を使い、Flag に適切な値を選ばせることが推奨されています。

バージョン 3.6 で追加.

IntFlag と同様に、Flag メンバーの組み合わせがどのフラグも設定されていない状態になった場合、その真偽値としての評価は False となります：

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

(次のページに続く)

(前のページからの続き)

```
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

個別のフラグは 2 のべき乗 (1, 2, 4, 8, ...) の値を持つべきですが、フラグの組み合わせはそうはなりません:

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

”フラグが設定されていない”状態に名前を付けても、その真偽値は変わりません:

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

Flag メンバーは反復処理することもできます:

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

バージョン 3.11 で追加.

---

**注釈:** ほとんどの新しいコードでは、Enum と Flag が強く推奨されます。というのは、IntEnum と IntFlag は (整数と比較でき、従って推移的に他の無関係な列挙型と比較できてしまうことにより) 列挙型の意味論的な約束に反するからです。IntEnum と IntFlag は、Enum や Flag では上手くいかない場合のみに使うべきです; 例えば、整数定数を列挙型で置き換えるときや、他のシステムとの相互運用性を持たせたいときです。

---

## 11.5 その他

`IntEnum` は `enum` モジュールの一部ですが、単独での実装もとても簡単に行なえます:

```
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`.

いくつかのルール:

1. When subclassing `Enum`, mix-in types must appear before `Enum` itself in the sequence of bases, as in the `IntEnum` example above.
2. 複合させる型はサブクラス化可能でなければいけません。例えば、`bool` と `range` はサブクラス化できないため、複合させると `Enum` 作成時にエラーが発生します。
3. `Enum` のメンバーはどんなデータ型でも構いませんが、追加のデータ型 (例えば、上の例の `int`) と複合させてしまうと、すべてのメンバーの値はそのデータ型でなければならなくなります。この制限は、メソッドの追加するだけの、他のデータ型を指定しない複合には適用されません。
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__()`.
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.

---

**注釈:** 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.

---

## 12 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.

例えば、複数の値をコンストラクタに渡すが、その中の 1 つだけを値として使いたい場合は次のようにします:

```

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

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

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

```

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

## 12.1 細かい点

### `__dunder__` 名のサポート

`__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.

### `_sunder_` 名のサポート

- `_name_` -- name of the member
- `_value_` -- value of the member; can be set / modified in `__new__`
- `_missing_` -- a lookup function used when a value is not found; may be overridden
- `_ignore_` -- a list of names, either as a `list` or a `str`, that will not be transformed into members, and will be removed from the final class

- `_order_` -- used in Python 2/3 code to ensure member order is consistent (class attribute, removed during class creation)
- `_generate_next_value_` -- used by the *Functional API* and by `auto` to get an appropriate value for an enum member; may be overridden

---

**注釈:** For standard Enum classes the next value chosen is the last value seen incremented by one.

For Flag classes the next value chosen will be the next highest power-of-two, regardless of the last value seen.

---

バージョン 3.6 で追加: `_missing_`, `_order_`, `_generate_next_value_`

バージョン 3.7 で追加: `_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']
```

---

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

---

## **`__Private__` names**

Private names は列挙型メンバーには変換されず、通常の属性となります。

バージョン 3.11 で変更.

## Enum メンバー型

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.

バージョン 3.5 で変更.

## 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
```

## Enum クラスとメンバーの真偽値

(`int`, `str` などのような) 非 Enum 型と複合させた enum クラスは、その複合された型の規則に従って評価されます; そうでない場合は、全てのメンバーは `True` と評価されます。メンバーの値に依存する独自の enum の真偽値評価を行うには、クラスに次のコードを追加してください:

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

プレーンな Enum クラスは `True` として評価されます。

## メソッド付きの Enum クラス

enum サブクラスに追加のメソッドを与えた場合、後述の *Planet* クラスのように、そのメソッドはメンバーの `dir()` に表示されますが、クラスの `dir()` には表示されません:

```
>>> 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']
```

## Flag のメンバーの組み合わせ

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

例として以下のスニペットを使用します:

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

the following are true:

- 単一ビットのフラグは正規形です
- 複数ビットや 0 ビットのフラグはエイリアスです
- 反復処理では正規形のフラグのみ返却されます:

```
>>> 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>
```

(次のページに続く)

(前のページからの続き)

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

- 名前のないフラグについては、そのメンバーの名前から名前が生成されます:

```
>>> (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>
```

- メンバーシップ/包含 のチェックでは、値が 0 のフラグは常に含まれるものとして扱われます:

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

それ以外では、一方のフラグの全ビットが他方のフラグに含まれる場合のみ、True が返されます:

```
>>> 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** --> 無効な値が指定された場合に例外を発生させる
- **CONFORM** --> 無効なビットを破棄する
- **EJECT** --> フラグのステータスを失い、指定された値を持つ通常の int となります。
- **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).

## 13 Enum と Flag はどう違うのか？

Enum は `Enum` 派生クラスやそれらのインスタンス (メンバー) 双方の多くの側面に影響を及ぼすカスタムメタクラスを持っています。

### 13.1 Enum クラス

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__()`).

### 13.2 Flag クラス

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.

### 13.3 Enum メンバー (インスタンス)

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.

### 13.4 Flag メンバー

フラグのメンバーは、`Flag` クラスと同様に反復処理することができ、正規のメンバーのみが返されます。例えば:

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

(BLACK、PURPLE、WHITE は表示されないことに注意。)

フラグのメンバーを反転させると、負の値ではなく、対応する正の値が返されます:

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

フラグのメンバーは、それが含む 2 のべき乗の値の数に対応する `length` を持ちます。例えば:

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

## 14 Enum クックブック

`Enum`, `IntEnum`, `StrEnum`, `Flag`, `IntFlag` は用途の大部分をカバーすると予想されますが、そのすべてをカバーできているわけではありません。ここでは、そのまま、あるいは独自の列挙型を作る例として使える、様々なタイプの列挙型を紹介します。

### 14.1 値の省略

多くの用途では、列挙型の実際の値が何かは気にされません。このタイプの単純な列挙型を定義する方法はいくつかあります:

- 値に `auto` インスタンスを使用する
- 値として `object` インスタンスを使用する
- 値として解説文字列を使用する
- use a tuple as the value and a custom `__new__()` to replace the tuple with an `int` value

これらのどの方法を使ってもユーザーに対して、値は重要ではなく、他のメンバーの番号の振り直しをする必要無しに、メンバーの追加、削除、並べ替えが行えることを示せます。

#### `auto` を使う

`auto` を使うと次のようになります:

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

## object を使う

object を使うと次のようになります:

```
>>> 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>
```

## 解説文字列を使う

値として文字列を使うと次のようになります:

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

## Using a custom `__new__()`

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
```

(次のページに続く)

(前のページからの続き)

```
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN: 2>
```

AutoNumber をより広い用途で使うには、シグニチャに `*args` を追加します:

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

AutoNumber を継承すると、追加の引数を取り扱える独自の `__init__` が書けます。

```
>>> 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'
```

---

**注釈:** 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.

---

**警告:** 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)
```

## 14.2 OrderedEnum

IntEnum をベースとしないため、通常の Enum の不変条件 (他の列挙型と比較できないなど) のままで、メンバーを順序付けできる列挙型です:

```
>>> 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
```

## 14.3 DuplicateFreeEnum

値が重複するメンバーがある場合に、エイリアスを作成するのではなくエラーを発生させます:

```
>>> 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))
```

(次のページに続く)

```

...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum: 'GRENE' --> 'GREEN'

```

---

**注釈:** これは Enum に別名を無効にするのと同様な振る舞いの追加や変更をおこなうためのサブクラス化に役立つ例です。単に別名を無効にしたいだけなら、`unique()` デコレーターを使用して行えます。

---

## 14.4 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

```

## 14.5 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)>]
```

## 15 EnumType のサブクラスを作る

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.