
Guide sur l'utilisation d'Enum

Version 3.13.3

Guido van Rossum and the Python development team

avril 25, 2025

Python Software Foundation
Email : docs@python.org

Table des matières

| | | |
|-----------|--|-----------|
| 1 | Accès par programme aux membres de l'énumération et à leurs attributs | 5 |
| 2 | Duplication des membres et des valeurs d'une énumération | 5 |
| 3 | Garantie de valeurs d'énumération uniques | 6 |
| 4 | Utilisation de valeurs automatiques | 6 |
| 5 | Itération | 7 |
| 6 | Comparaisons | 7 |
| 7 | Membres et attributs autorisés des énumérations | 8 |
| 8 | Restrictions sur la dérivation d'énumérations | 9 |
| 9 | Dataclass support | 9 |
| 10 | Sérialisation | 10 |
| 11 | API fonctionnelle | 11 |
| 12 | Déclinaisons d'énumérations | 12 |
| 12.1 | <i>IntEnum</i> | 12 |
| 12.2 | <i>StrEnum</i> | 13 |
| 12.3 | <i>IntFlag</i> | 13 |
| 12.4 | <i>Flag</i> | 15 |
| 12.5 | Autres énumérations | 16 |
| 13 | When to use <code>__new__()</code> vs. <code>__init__()</code> | 16 |
| 13.1 | Approfondissements | 17 |
| 14 | How are Enums and Flags different? | 20 |
| 14.1 | Classes <i>Enum</i> | 20 |
| 14.2 | Flag Classes | 21 |
| 14.3 | Membres d'une <i>Enum</i> (les instances) | 21 |
| 14.4 | Flag Members | 21 |

| | |
|----------------------------------|-----------|
| 15 Enum Cookbook | 21 |
| 15.1 Omission de valeurs | 21 |
| 15.2 Énumération ordonnée | 23 |
| 15.3 Énumération sans doublon | 24 |
| 15.4 MultiValueEnum | 24 |
| 15.5 Planète | 25 |
| 15.6 Intervalle de temps | 25 |
| 16 Dérivations d'EnumType | 26 |

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.

Elles sont particulièrement utiles lorsque vous avez une variable qui peut prendre une valeur dans une plage limitée de valeurs. Par exemple, les jours de la semaine :

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

Ou alors les couleurs primaires RVB (NdT : RGB en anglais) :

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

Comme vous pouvez le voir, créer une Enum est très simple à écrire, il suffit qu'elle hérite de Enum elle-même.

Note

casse des membres d'une énumération

Because Enums are used to represent constants, and to help avoid issues with name clashes between mixin-class methods/attributes and enum names, we strongly recommend using UPPER_CASE names for members, and will be using that style in our examples.

Selon la nature de l'énumération, il peut être important d'avoir accès à la valeur d'un membre. Dans tous les cas, cette valeur peut être utilisée pour obtenir le membre correspondant :

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

Comme vous pouvez le constater, la `repr()` d'un membre affiche le nom de l'énumération, le nom du membre et la valeur. La `str()` d'un membre affiche uniquement le nom de l'énumération et le nom du membre :

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

Le type d'un membre d'énumération est l'énumération à laquelle il appartient :

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

Maintenant, nous pouvons découvrir quel jour nous sommes aujourd'hui ! Regardez :

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

Bien sûr, si vous lisez cette page un autre jour, vous verrez ce jour-là à la place.

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
```

(suite sur la page suivante)

```
... SATURDAY = 32
... SUNDAY = 64
```

Nous avons changé deux choses : nous héritons de `Flag` et les valeurs sont toutes des puissances de 2.

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

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

Mais `Flag` nous permet aussi de combiner plusieurs membres en une seule variable :

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

Nous pouvons même itérer sur une variable `Flag` :

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

Bien, préparons quelques corvées :

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

Et une fonction pour afficher les corvées d'un jour donné :

```
>>> 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 Accès par programme aux membres de l'énumération et à leurs attributs

Il est parfois utile d'accéder aux membres d'une énumération programmatiquement (c'est-à-dire les cas où `Color.RED` ne suffit pas car la couleur exacte n'est pas connue au moment de l'écriture du programme). `Enum` permet de tels accès :

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

Si vous souhaitez accéder aux membres d'une énumération par leur nom, utilisez l'accès par indice :

```
>>> 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 Duplication des membres et des valeurs d'une énumération

Il n'est pas licite d'avoir deux membres d'une énumération avec le même nom :

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

Cependant, un membre peut avoir d'autres noms qui lui sont associés. Étant donné deux entrées `A` et `B` avec la même valeur (et `A` définie en premier), `B` est un synonyme (ou alias) du membre `A`. La recherche par valeur de la valeur de `A` renvoie le membre `A`. La recherche par nom de `A` renvoie le membre `A`. La recherche par nom de `B` renvoie également le membre `A` :

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

Note

tenter de créer un membre portant le même nom qu'un attribut déjà défini (un autre membre, une méthode, etc.) ou tenter de créer un attribut portant le même nom qu'un membre n'est pas autorisé.

3 Garantie de valeurs d'énumération uniques

Par défaut, les énumérations autorisent plusieurs noms comme synonymes pour la même valeur. Lorsque ce comportement n'est pas souhaité, vous pouvez utiliser le décorateur `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 Utilisation de valeurs automatiques

Si la valeur exacte n'est pas importante, vous pouvez utiliser `auto` :

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

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

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

Note

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

5 Itération

L'itération sur les membres d'une énumération ne fournit pas les synonymes :

```
>>> 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 that the aliases `Shape.ALIAS_FOR_SQUARE` and `Weekday.WEEKEND` aren't shown.

L'attribut spécial `__members__` est un tableau de correspondances ordonné en lecture seule des noms vers les membres. Il inclut tous les noms définis dans l'énumération, y compris les synonymes :

```
>>> 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>)
```

L'attribut `__members__` peut être utilisé pour un accès programmatique détaillé aux membres de l'énumération. Par exemple, trouver tous les synonymes :

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

Note

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

6 Comparaisons

Les membres de l'énumération sont comparés par identité :

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

Les comparaisons ordonnées entre les valeurs d'énumération *ne sont pas* prises en charge. Les membres d'une énumération ne sont pas des entiers (mais voir *IntEnum* plus bas) :

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

Les comparaisons d'égalité sont cependant définies :

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

(suite sur la page suivante)

```
True
>>> Color.BLUE == Color.BLUE
True
```

Les comparaisons avec des valeurs en dehors de l'énumération sont toujours fausses (encore une fois, `IntEnum` a été explicitement conçue pour se comporter différemment, voir ci-dessous) :

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

Avertissement

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

7 Membres et attributs autorisés des énumérations

La plupart des exemples ci-dessus utilisent des nombres entiers pour les valeurs d'énumération. L'utilisation d'entiers est rapide et pratique (et fournie par défaut par l'*API fonctionnelle*), mais n'est pas strictement obligatoire. Dans la grande majorité des cas, on ne se soucie pas de la valeur réelle des membres d'une énumération. Mais si la valeur *est* importante, les énumérations peuvent avoir des valeurs arbitraires.

Les énumérations sont des classes Python et peuvent avoir des méthodes ainsi que des méthodes spéciales comme d'habitude. Prenons cette énumération :

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

alors :

```
>>> 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__`.


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

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

Modifié dans la version 3.12 : Only the dataclass fields are shown in the value area, not the dataclass' name.

Note

Adding `dataclass()` decorator to `Enum` and its subclasses is not supported. It will not raise any errors, but it will produce very strange results at runtime, such as members being equal to each other :

```
>>> @dataclass                                # don't do this: it does not make any sense
... class Color(Enum):
...     RED = 1
...     BLUE = 2
...
>>> Color.RED is Color.BLUE
False
>>> Color.RED == Color.BLUE # problem is here: they should not be equal
True
```

10 Sérialisation

Les énumérations peuvent être sérialisées et désérialisées :

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

Les restrictions habituelles pour la sérialisation s'appliquent : les énumérations sérialisables doivent être définies au niveau supérieur d'un module, car la sérialisation nécessite qu'elles puissent être importées à partir de ce module.

Note

avec la version 4 du protocole *pickle*, il est possible de sérialiser facilement les énumérations imbriquées dans d'autres classes.

It is possible to modify how enum members are pickled/unpickled by defining `__reduce_ex__()` in the enumeration class. The default method is `by-value`, but enums with complicated values may want to use `by-name` :

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

Note

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

11 API fonctionnelle

La classe `Enum` est callable, elle fournit l'API fonctionnelle suivante :

```
>>> 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 sémantique de cette API ressemble à celle des `namedtuple`. Le premier argument de l'appel à `Enum` est le nom de l'énumération.

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
... 
```

La raison de la valeur par défaut de 1 comme numéro de départ et non de 0 est que le sens booléen de 0 est `False`. Or, par défaut, les membres d'une énumération s'évaluent tous à `True`.

La sérialisation d'énumérations créées avec l'API fonctionnelle peut être délicat car les détails d'implémentation de la pile d'appels sont utilisés pour essayer de déterminer dans quel module l'énumération est créée (par exemple, cela échoue si vous utilisez une fonction utilitaire dans un module séparé, et peut également ne pas fonctionner sur *IronPython* ou *Jython*). La solution consiste à spécifier explicitement le nom du module comme suit :

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

Avertissement

si le `module` n'est pas fourni et qu'`Enum` ne peut pas le déterminer, les nouveaux membres de l'énumération ne pourront pas être sélectionnés ; pour garder les erreurs au plus près de la source, la sérialisation est désactivée.

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')
```

La signature complète est :

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

(suite sur la page suivante)

```
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'
```

ou un itérateur sur des noms :

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

ou un itérateur sur des paires (nom, valeur) :

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

ou un tableau de correspondances :

```
{'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.

Modifié dans la version 3.5 : le paramètre *start* a été ajouté.

12 Déclinaisons d'énumérations

12.1 IntEnum

La première variante de `Enum` fournie est également une sous-classe de `int`. Les membres d'une `IntEnum` peuvent être comparés à des entiers ; par extension, des énumérations `IntEnum` de différents types peuvent aussi être comparées :

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

Cependant, elles ne peuvent toujours pas être comparées aux énumérations classiques `Enum` :

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

(suite sur la page suivante)

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

Les valeurs `IntEnum` se comportent comme des entiers, comme vous pouvez vous y attendre :

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

12.2 *StrEnum*

La deuxième variante de `Enum` fournie est également une sous-classe de `str`. Les membres d'une `StrEnum` peuvent être comparés à des chaînes ; par extension, les énumérations *StrEnum* de différents types peuvent également être comparées les unes aux autres.

Ajouté dans la version 3.11.

12.3 *IntFlag*

La variante suivante fournie d'`Enum` est `IntFlag`, également basée sur `int`. La différence étant que les membres `IntFlag` peuvent être combinés à l'aide des opérateurs bit-à-bit (`&`, `|`, `^`, `~`) et le résultat est toujours un membre `IntFlag`, si possible. Comme `IntEnum`, les membres de `IntFlag` sont aussi des entiers et peuvent être utilisés partout où on utilise un `int`.

Note

toute opération sur un membre `IntFlag`, en dehors des opérations bit-à-bit, fait perdre l'appartenance à `IntFlag`.

Les opérations bit-à-bit qui entraînent des valeurs `IntFlag` invalides font perdre l'appartenance à `IntFlag`. Voir `FlagBoundary` pour plus de détails.

Ajouté dans la version 3.6.

Modifié dans la version 3.11.

Classe exemple dérivée d'`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
```

Il est également possible de nommer les combinaisons :

```
>>> 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>
```

Note

les combinaisons nommées sont considérées être des synonymes (ou alias). Les synonymes n'apparaissent pas dans une itération, mais peuvent être renvoyés à partir de recherches par valeur.

Modifié dans la version 3.11.

Une autre différence importante entre `IntFlag` et `Enum` est que si aucun indicateur n'est défini (la valeur est 0), son évaluation booléenne est `False` :

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

Comme les membres d'une `IntFlag` sont aussi des sous-classes de `int`, ils peuvent être combinés avec eux (mais peuvent alors perdre l'appartenance à `IntFlag`) :

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

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

Note

l'opérateur de négation, `~`, renvoie toujours un membre `IntFlag` avec une valeur positive :

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

On peut aussi itérer sur les membres d'une `IntFlag` :

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

Ajouté dans la version 3.11.

12.4 Flag

La dernière variante est `Flag`. Comme `IntFlag`, les membres de `Flag` peuvent être combinés à l'aide des opérateurs bit-à-bit (`&`, `|`, `^`, `~`). Contrairement à `IntFlag`, ils ne peuvent être combinés ni comparés à aucune autre énumération `Flag`, ni à `int`. Bien qu'il soit possible de spécifier les valeurs directement, il est recommandé d'utiliser `auto` comme valeur et de laisser `Flag` sélectionner une valeur appropriée.

Ajouté dans la version 3.6.

Comme pour `IntFlag`, si une combinaison de membres `Flag` entraîne qu'aucun indicateur n'est défini, l'évaluation booléenne est `False` :

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

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

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

Donner un nom à la condition « aucun indicateur défini » ne change pas sa valeur booléenne :

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

On peut aussi itérer sur les membres d'une `Flag` :

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

Ajouté dans la version 3.11.

Note

pour la majorité du code nouveau, `Enum` et `Flag` sont fortement recommandées, car `IntEnum` et `IntFlag` brisent certaines promesses sémantiques d'une énumération (en pouvant être comparées à des entiers et donc,

par transitivité, à d'autres énumérations sans rapport). `IntEnum` et `IntFlag` ne doivent être utilisées que dans les cas où `Enum` et `Flag` ne suffisent pas ; par exemple, lorsque des constantes entières sont remplacées par des énumérations, ou pour l'interopérabilité avec d'autres systèmes.

12.5 Autres énumérations

Bien que `IntEnum` fasse partie du module `enum`, il serait très simple de l'implémenter indépendamment :

```
class IntEnum(int, ReprEnum):    # or Enum instead of ReprEnum
    pass
```

This demonstrates how similar derived enumerations can be defined ; for example a `FloatEnum` that mixes in `float` instead of `int`.

Quelques règles :

1. When subclassing `Enum`, mix-in types must appear before the `Enum` class itself in the sequence of bases, as in the `IntEnum` example above.
2. Les types mélangés doivent pouvoir être sous-classés. Par exemple, `bool` et `range` ne peuvent pas être sous-classés et génèrent une erreur lors de la création d'une énumération s'ils sont utilisés comme type de mélange.
3. Alors qu'`Enum` peut avoir des membres de n'importe quel type, une fois que vous avez mélangé un type supplémentaire, tous les membres doivent avoir des valeurs de ce type, par exemple `int` ci-dessus. Cette restriction ne s'applique pas aux classes de mélange qui ajoutent uniquement des méthodes et ne spécifient pas d'autre type.
4. When another data type is mixed in, the `value` attribute is *not the same* as the enum member itself, although it is equivalent and will compare equal.
5. A data type is a mixin that defines `__new__()`, or a `dataclass`
6. %-style formatting : `%s` and `%r` call the `Enum` class's `__str__()` and `__repr__()` respectively ; other codes (such as `%i` or `%h` for `IntEnum`) treat the enum member as its mixed-in type.
7. Formatted string literals, `str.format()`, and `format()` will use the enum's `__str__()` method.

Note

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

13 When to use `__new__()` VS. `__init__()`

`__new__()` must be used whenever you want to customize the actual value of the `Enum` member. Any other modifications may go in either `__new__()` or `__init__()`, with `__init__()` being preferred.

Par exemple, si vous souhaitez passer plusieurs éléments au constructeur, mais que vous souhaitez qu'un seul d'entre eux soit la valeur :

```
>>> 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')
```

(suite sur la page suivante)


```

...     VX = (2, 'V.X', 'km/s')
...     VY = (3, 'V.Y', 'km/s')
...

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

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

```

⚠ Avertissement

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

13.1 Approfondissements

Noms de la forme `__dunder__` disponibles

`__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.

Noms de la forme `_sunder_` disponibles

- `_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
- `_generate_next_value_()` -- used to get an appropriate value for an enum member; may be overridden
- `_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. See [MultiValueEnum](#) for an example.

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.

Modifié dans la version 3.13 : Prior versions would use the last seen value instead of the highest value.

Ajouté dans la version 3.6 : `_missing_`, `_order_`, `_generate_next_value_`

Ajouté dans la version 3.7 : `_ignore_`

Ajouté dans la version 3.13 : `_add_alias_`, `_add_value_alias_`

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']
```

Note

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

__Private__ names

Les noms privés ne sont pas convertis en membres de l'énumération, mais restent des attributs normaux.

Modifié dans la version 3.11.

Types des membres d'une Enum

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

Modifié dans la version 3.5.

Création de membres mélangés avec d'autres types de données

Lorsque vous dérivez d'autres types de données, tels que `int` ou `str`, avec une Enum, toutes les valeurs après `=` sont passées au constructeur de ce type de données. Par exemple :

```
>>> 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
```

Valeur booléenne des classes et membres Enum

Les classes Enum mélangées avec des types non-Enum (tels que `int`, `str`, etc.) sont évaluées selon les règles du type mélangé; sinon, tous les membres sont évalués comme `True`. Pour que l'évaluation booléenne de votre propre énumération dépende de la valeur du membre, ajoutez ce qui suit à votre classe :

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

Les classes simples Enum sont toujours évaluées comme `True`.

Méthodes dans les classes Enum

Si vous dotez votre sous-classe énumération de méthodes supplémentaires, comme la classe *Planet* ci-dessous, ces méthodes apparaissent dans le `dir()` des membres, mais pas dans celui de la classe :

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__class__', '__doc__', '__members__', '__module__']
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'mass', 'name', 'radius', 'surface_gravity',
↪ 'value']
```

Combinaisons de membres de `Flag`

L'itération sur une combinaison de membres `Flag` ne renvoie que les membres dont un seul bit est à 1 :

```
>>> 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>
```

Précisions sur Flag et IntFlag

En utilisant l'extrait suivant pour nos exemples :

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

ce qui suit est vrai :

- les membres dont un seul bit est à 1 sont canoniques ;
- ceux qui ont plusieurs bits à 1 ou aucun bit à 1 sont des synonymes ;
- seuls les membres canoniques sont renvoyés pendant une itération :

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

- la négation d'un membre ou d'une composition de membres renvoie un nouveau membre (ou composition de membres) avec la valeur entière positive correspondante :

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

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

- les noms des pseudo-membres sont construits à partir des noms de leurs membres :

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

>>> class Perm(IntFlag):
...     R = 4
```

(suite sur la page suivante)

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

- les membres avec plusieurs bits à 1 (des alias) peuvent être renvoyés par les opérations :

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

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

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

- pour la vérification d'appartenance, les membres dont la valeur est 0 sont toujours considérés comme étant membres :

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

sinon, seulement si tous les bits à 1 d'un membre sont aussi à 1 dans l'autre membre, renvoie *True* :

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

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

Il existe un nouveau mécanisme de délimitation qui contrôle la façon dont les bits hors plage/invalides sont gérés : *STRICT*, *CONFORM*, *EJECT* et *KEEP* :

- *STRICT* --> lève une exception lorsqu'on lui présente des valeurs invalides
- *CONFORM* --> ignore les bits invalides
- *EJECT* --> la valeur présentée perd le statut de membre et devient un entier normal
- *KEEP* --> garde les bits supplémentaires
 - garde le statut de membre avec les bits supplémentaires
 - les bits supplémentaires ne sont pas produits dans une itération
 - les bits supplémentaires ne sont pas représentés par *repr()* et *str()*

La valeur par défaut pour *Flag* est *STRICT*, la valeur par défaut pour *IntFlag* est *EJECT* et la valeur par défaut pour *_convert_* est *KEEP* (voir *ssl.Options* pour un exemple de cas où *KEEP* est nécessaire).

14 How are Enums and Flags different ?

Les énumérations ont une métaclasse personnalisée qui modifie de nombreux aspects des classes dérivées d'*Enum* et de leurs instances (membres).

14.1 Classes *Enum*

The *EnumType* metaclass is responsible for providing the *__contains__()*, *__dir__()*, *__iter__()* and other methods that allow one to do things with an *Enum* class that fail on a typical class, such as *list(Color)* or *some_enum_var in Color*. *EnumType* is responsible for ensuring that various other methods on the final *Enum* class are correct (such as *__new__()*, *__getnewargs__()*, *__str__()* and *__repr__()*).

14.2 Flag Classes

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

14.3 Membres d'une *Enum* (les instances)

The most interesting thing about enum members is that they are singletons. `EnumType` creates them all while it is creating the enum class itself, and then puts a custom `__new__()` in place to ensure that no new ones are ever instantiated by returning only the existing member instances.

14.4 Flag Members

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

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

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

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

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

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

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

15 Enum Cookbook

`Enum`, `IntEnum`, `StrEnum`, `Flag` et `IntFlag` sont censées couvrir la majorité des cas d'utilisation, mais elles ne peuvent pas tout couvrir. Voici quelques recettes pour différents types d'énumérations qui peuvent être utilisées directement, ou pour servir d'exemples afin de créer les vôtres.

15.1 Omission de valeurs

Dans de nombreux cas d'utilisation, on ne se soucie pas de la valeur réelle d'une énumération. Il existe plusieurs manières de définir ce type d'énumération simple :

- utilisez des instances d'`auto` pour la valeur ;
- utilisez des instances d'`object` comme valeur ;
- utilisez une chaîne de caractères descriptive comme valeur ;
- use a tuple as the value and a custom `__new__()` to replace the tuple with an `int` value

L'utilisation de l'une de ces méthodes signifie au lecteur que ces valeurs ne sont pas importantes et permet également d'ajouter, de supprimer ou de réorganiser des membres sans avoir à renuméroter les autres membres.

Utilisation d'`auto`

Voici un exemple où nous utilisons `auto` :

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

(suite sur la page suivante)

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

Utilisation d'object

Voici un exemple où nous utilisons `object` :

```
>>> 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>
```

Utilisation d'une chaîne de caractères descriptive

Voici un exemple où nous utilisons une chaîne de caractères :

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

Pour créer une `AutoNumber` plus générale, ajoutons `*args` à la signature :

```
>>> class AutoNumber(Enum):
...     def __new__(cls, *args):          # this is the only change from above
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
... 
```

Ensuite, lorsque nous héritons de `AutoNumber`, nous pouvons écrire notre propre `__init__` pour gérer les arguments supplémentaires :

```
>>> 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'
```

Note

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.

Avertissement

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

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

15.2 Énumération ordonnée

Voici une énumération ordonnée qui n'est pas basée sur `IntEnum` et maintient donc les invariants normaux d'Enum (comme ne pas pouvoir être comparée à d'autres énumérations) :

```
>>> 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
```

(suite sur la page suivante)

```

...         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 Énumération sans doublon

Raises an error if a duplicate member value is found instead of creating an alias :

```

>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum: %r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum: 'GRENE' --> 'GREEN'

```

Note

c'est un exemple utile de dérivation d'*Enum* pour ajouter ou modifier d'autres comportements ainsi que pour interdire les synonymes. Si le seul changement souhaité est l'interdiction des synonymes, le décorateur `unique()` peut être utilisé à la place.

15.4 MultiValueEnum

Supports having more than one value per member :

```

>>> class MultiValueEnum(Enum):
...     def __new__(cls, value, *values):
...         self = object.__new__(cls)
...         self._value_ = value

```

(suite sur la page suivante)


```

...         for v in values:
...             self._add_value_alias_(v)
...         return self
...
>>> class DType(MultiValueEnum):
...     float32 = 'f', 8
...     double64 = 'd', 9
...
>>> DType('f')
<DType.float32: 'f'>
>>> DType(9)
<DType.double64: 'd'>

```

15.5 Planète

If `__new__()` or `__init__()` is defined, the value of the enum member will be passed to those methods :

```

>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27, 7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
...         self.mass = mass          # in kilograms
...         self.radius = radius      # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129

```

15.6 Intervalle de temps

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:]

```

(suite sur la page suivante)

(suite de la page précédente)

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

16 Dérivations d'*EnumType*

Alors que la plupart des besoins d'énumérations peuvent être satisfaits en sous-classant `Enum`, avec des décorateurs de classe ou des fonctions personnalisées, `EnumType` peut être dérivée pour créer des énumérations vraiment différentes.