

---

# Enum HOWTO

Release 3.10.0a6

Guido van Rossum  
and the Python development team

April 04, 2021

Python Software Foundation  
Email: docs@python.org

## Contents

<b>1</b>	<b>Basic Enum Tutorial</b>	<b>2</b>
<b>2</b>	<b>Programmatic access to enumeration members and their attributes</b>	<b>4</b>
<b>3</b>	<b>Duplicating enum members and values</b>	<b>5</b>
<b>4</b>	<b>Ensuring unique enumeration values</b>	<b>5</b>
<b>5</b>	<b>Using automatic values</b>	<b>6</b>
<b>6</b>	<b>Iteration</b>	<b>6</b>
<b>7</b>	<b>Comparisons</b>	<b>7</b>
<b>8</b>	<b>Allowed members and attributes of enumerations</b>	<b>7</b>
<b>9</b>	<b>Restricted Enum subclassing</b>	<b>8</b>
<b>10</b>	<b>Pickling</b>	<b>8</b>
<b>11</b>	<b>Functional API</b>	<b>9</b>
<b>12</b>	<b>Derived Enumerations</b>	<b>10</b>
12.1	IntEnum . . . . .	10
12.2	StrEnum . . . . .	11
12.3	IntFlag . . . . .	11
12.4	Flag . . . . .	13
12.5	Others . . . . .	14
<b>13</b>	<b>When to use <code>__new__()</code> vs. <code>__init__()</code></b>	<b>14</b>
13.1	Finer Points . . . . .	15
<b>14</b>	<b>How are Enums different?</b>	<b>19</b>
14.1	Enum Classes . . . . .	19
14.2	Enum Members (aka instances) . . . . .	19
14.3	Omitting values . . . . .	19
14.4	OrderedEnum . . . . .	21
14.5	DuplicateFreeEnum . . . . .	22
14.6	Planet . . . . .	22

14.7 TimePeriod . . . . .	23
14.8 Conforming input to Flag . . . . .	23

## 15 Subclassing EnumType 23

---

**Author** Ethan Furman <ethan at stoneleaf dot us>

# 1 Basic Enum Tutorial

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.

They are most useful when you have a variable that can take one of a limited selection of values. For example, the days of the week:

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

As you can see, creating an Enum is as simple as writing a class that inherits from Enum itself.

---

### Note: Case of Enum Members

Because Enums are used to represent constants we recommend using UPPER\_CASE names for members, and will be using that style in our examples.

---

Depending on the nature of the enum a member's value may or may not be important, but either way that value can be used to get the corresponding member:

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

As you can see, the `repr()` of a member shows the enum name and the member name. The `str()` on a member shows only its name:

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

The *type* of an enumeration member is the enum it belongs to:

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

Now we can find out what today is! Observe:

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

Of course, if you're reading this on some other day, you'll see that day instead.

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
```

We've changed two things: we're inherited from `Flag`, and the values are all powers of 2.

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

```
>>> first_week_day = Weekday.MONDAY
>>> first_week_day
Weekday.MONDAY
```

But `Flag` also allows us to combine several members into a single variable:

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

You can even iterate over a Flag variable:

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

Okay, let's get some chores set up:

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

And a function to display the chores for a given day:

```
>>> 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()
```

## 2 Programmatic access to enumeration members and their attributes

Sometimes it's useful to access members in enumerations programmatically (i.e. situations where `Color.RED` won't do because the exact color is not known at program-writing time). Enum allows such access:

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

If you want to access enum members by *name*, use item access:

```
>>> Color['RED']
Color.RED
>>> Color['GREEN']
Color.GREEN
```

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

```
>>> member = Color.RED
>>> member.name
```

(continues on next page)

```
'RED'
>>> member.value
1
```

### 3 Duplicating enum members and values

Having two enum members with the same name is invalid:

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

However, an enum member can have other names associated with it. Given two entries A and B with the same value (and A defined first), B is an alias for the member A. By-value lookup of the value of A will return the member A. By-name lookup of A will return the member A. By-name lookup of B will also return the member A:

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

---

**Note:** Attempting to create a member with the same name as an already defined attribute (another member, a method, etc.) or attempting to create an attribute with the same name as a member is not allowed.

---

### 4 Ensuring unique enumeration values

By default, enumerations allow multiple names as aliases for the same value. When this behavior isn't desired, you can use the `unique()` decorator:

```
>>> 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
```

## 5 Using automatic values

If the exact value is unimportant you can use `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):
...     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 Color]
['NORTH', 'SOUTH', 'EAST', 'WEST']
```

---

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

---

## 6 Iteration

Iterating over the members of an enum does not provide the aliases:

```
>>> list(Shape)
[Shape.SQUARE, Shape.DIAMOND, Shape.CIRCLE]
```

The special attribute `__members__` is a read-only ordered mapping of names to members. It includes all names defined in the enumeration, including the aliases:

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', Shape.SQUARE)
('DIAMOND', Shape.DIAMOND)
('CIRCLE', Shape.CIRCLE)
('ALIAS_FOR_SQUARE', Shape.SQUARE)
```

The `__members__` attribute can be used for detailed programmatic access to the enumeration members. For example, finding all the aliases:

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

## 7 Comparisons

Enumeration members are compared by identity:

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

Ordered comparisons between enumeration values are *not* supported. Enum members are not integers (but see *IntEnum* below):

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

Equality comparisons are defined though:

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

Comparisons against non-enumeration values will always compare not equal (again, *IntEnum* was explicitly designed to behave differently, see below):

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

## 8 Allowed members and attributes of enumerations

Most of the examples above use integers for enumeration values. Using integers is short and handy (and provided by default by the *Functional API*), but not strictly enforced. In the vast majority of use-cases, one doesn't care what the actual value of an enumeration is. But if the value *is* important, enumerations can have arbitrary values.

Enumerations are Python classes, and can have methods and special methods as usual. If we have this enumeration:

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # self is the member here
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
... 
```

Then:



---

**Note:** With pickle protocol version 4 it is possible to easily pickle enums nested in other classes.

---

It is possible to modify how enum members are pickled/unpickled by defining `__reduce_ex__()` in the enumeration class.

## 11 Functional API

The Enum class is callable, providing the following functional API:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
Animal.ANT
>>> Animal.ANT.value
1
>>> list(Animal)
[Animal.ANT, Animal.BEE, Animal.CAT, Animal.DOG]
```

The semantics of this API resemble `namedtuple`. The first argument of the call to `Enum` is the name of the enumeration.

The second argument is the *source* of enumeration member names. It can be a whitespace-separated string of names, a sequence of names, a sequence of 2-tuples with key/value pairs, or a mapping (e.g. dictionary) of names to values. The last two options enable assigning arbitrary values to enumerations; the others auto-assign increasing integers starting with 1 (use the `start` parameter to specify a different starting value). A new class derived from `Enum` is returned. In other words, the above assignment to `Animal` is equivalent to:

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

The reason for defaulting to 1 as the starting number and not 0 is that 0 is `False` in a boolean sense, but by default enum members all evaluate to `True`.

Pickling enums created with the functional API can be tricky as frame stack implementation details are used to try and figure out which module the enumeration is being created in (e.g. it will fail if you use a utility function in separate module, and also may not work on IronPython or Jython). The solution is to specify the module name explicitly as follows:

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

**Warning:** If `module` is not supplied, and `Enum` cannot determine what it is, the new `Enum` members will not be unpicklable; to keep errors closer to the source, pickling will be disabled.

The new pickle protocol 4 also, in some circumstances, relies on `__qualname__` being set to the location where pickle will be able to find the class. For example, if the class was made available in class `SomeData` in the global scope:

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

The complete signature is:

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

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

**names** The enum members. This can be a whitespace or comma separated string (values will start at 1 unless otherwise specified):

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

or an iterator of names:

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

or an iterator of (name, value) pairs:

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

or a mapping:

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

**module** name of module where new enum class can be found.

**qualname** where in module new enum class can be found.

**type** type to mix in to new enum class.

**start** number to start counting at if only names are passed in.

Changed in version 3.5: The *start* parameter was added.

## 12 Derived Enumerations

### 12.1 IntEnum

The first variation of Enum that is provided is also a subclass of `int`. Members of an `IntEnum` can be compared to integers; by extension, integer enumerations of different types can also be compared to each other:

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

However, they still can't be compared to standard Enum enumerations:

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

IntEnum values behave like integers in other ways you'd expect:

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

## 12.2 StrEnum

The second variation of Enum that is provided is also a subclass of `str`. Members of a `StrEnum` can be compared to strings; by extension, string enumerations of different types can also be compared to each other. `StrEnum` exists to help avoid the problem of getting an incorrect member:

```
>>> from enum import StrEnum
>>> class Directions(StrEnum):
...     NORTH = 'north', # notice the trailing comma
...     SOUTH = 'south'
```

Before `StrEnum`, `Directions.NORTH` would have been the tuple `('north',)`.

New in version 3.10.

## 12.3 IntFlag

The next variation of Enum provided, `IntFlag`, is also based on `int`. The difference being `IntFlag` members can be combined using the bitwise operators (`&`, `|`, `^`, `~`) and the result is still an `IntFlag` member, if possible. However, as the name implies, `IntFlag` members also subclass `int` and can be used wherever an `int` is used.

---

**Note:** Any operation on an `IntFlag` member besides the bit-wise operations will lose the `IntFlag` membership. Bit-wise operations that result in invalid `IntFlag` values will lose the `IntFlag` membership. See `FlagBoundary` for details.

---

New in version 3.6.

Changed in version 3.10.

Sample `IntFlag` class:

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
```

(continues on next page)

(continued from previous page)

```
...
>>> Perm.R | Perm.W
Perm.R|Perm.W
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

It is also possible to name the combinations:

```
>>> class Perm(IntFlag) :
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
Perm.RWX
>>> ~Perm.RWX
Perm(0)
>>> Perm(7)
Perm.RWX
```

---

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

---

Changed in version 3.10.

Another important difference between `IntFlag` and `Enum` is that if no flags are set (the value is 0), its boolean evaluation is `False`:

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

Because `IntFlag` members are also subclasses of `int` they can be combined with them (but may lose `IntFlag` membership):

```
>>> Perm.X | 4
Perm.R|Perm.X

>>> Perm.X | 8
9
```

---

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

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

---

`IntFlag` members can also be iterated over:

```
>>> list(RW)
[Perm.R, Perm.W]
```

New in version 3.10.

## 12.4 Flag

The last variation is `Flag`. Like `IntFlag`, `Flag` members can be combined using the bitwise operators (`&`, `|`, `^`, `~`). Unlike `IntFlag`, they cannot be combined with, nor compared against, any other `Flag` enumeration, nor `int`. While it is possible to specify the values directly it is recommended to use `auto` as the value and let `Flag` select an appropriate value.

New in version 3.6.

Like `IntFlag`, if a combination of `Flag` members results in no flags being set, the boolean evaluation is `False`:

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

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

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

Giving a name to the “no flags set” condition does not change its boolean value:

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

Flag members can also be iterated over:

```
>>> purple = Color.RED | Color.BLUE
>>> list(purple)
[Color.RED, Color.BLUE]
```

New in version 3.10.

---

**Note:** For the majority of new code, `Enum` and `Flag` are strongly recommended, since `IntEnum` and `IntFlag` break some semantic promises of an enumeration (by being comparable to integers, and thus by transitivity to other unrelated enumerations). `IntEnum` and `IntFlag` should be used only in cases where `Enum` and `Flag` will not do; for example, when integer constants are replaced with enumerations, or for interoperability with other systems.

---

## 12.5 Others

While `IntEnum` is part of the `enum` module, it would be very simple to implement independently:

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

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

Some rules:

1. When subclassing `Enum`, mix-in types must appear before `Enum` itself in the sequence of bases, as in the `IntEnum` example above.
2. While `Enum` can have members of any type, once you mix in an additional type, all the members must have values of that type, e.g. `int` above. This restriction does not apply to mix-ins which only add methods and don't specify another type.
3. 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.
4. %-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.
5. Formatted string literals, `str.format()`, and `format()` will use the mixed-in type's `__format__()` unless `__str__()` or `__format__()` is overridden in the subclass, in which case the overridden methods or `Enum` methods will be used. Use the `!s` and `!r` format codes to force usage of the `Enum` class's `__str__()` and `__repr__()` methods.

## 13 When to use `__new__()` vs. `__init__()`

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

For example, if you want to pass several items to the constructor, but only want one of them to be the value:

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

## 13.1 Finer Points

### 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

- `_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

---

**Note:** 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.

---

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

New in version 3.7: `_ignore_`

To help keep Python 2 / Python 3 code in sync an `_order_` attribute can be provided. It will be checked against the actual order of the enumeration and raise an error if the two do not match:

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_:
['RED', 'BLUE', 'GREEN']
['RED', 'GREEN', 'BLUE']
```

---

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

---

## `_Private__names`

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

Changed in version 3.10.

## Enum member type

Enum members are instances of their enum class, and are normally accessed as `EnumClass.member`. In Python versions 3.5 to 3.9 you could access members from other members – this practice was discouraged, and in 3.12 Enum will return to not allowing it, while in 3.10 and 3.11 it will raise a `DeprecationWarning`:

```
>>> class FieldTypes (Enum) :
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
DeprecationWarning: accessing one member from another is not supported,
    and will be disabled in 3.12
<FieldTypes.size: 2>
```

Changed in version 3.5.

Changed in version 3.10.

## 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) :
...     example = '11', 16      # '11' will be interpreted as a hexadecimal
...                             # number
>>> MyEnum.example.value
17
```

## Boolean value of Enum classes and members

Enum classes that are mixed with non-Enum types (such as `int`, `str`, etc.) are evaluated according to the mixed-in type's rules; otherwise, all members evaluate as `True`. To make your own enum's boolean evaluation depend on the member's value add the following to your class:

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

Plain Enum classes always evaluate as `True`.



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

```
>>> Color.BLUE
Color.BLUE

>>> ~Color.BLUE
Color.RED|Color.GREEN
```

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

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

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

```
>>> Color.RED | Color.BLUE
Color.PURPLE

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

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

- membership / containment checking has changed slightly – zero valued flags are never considered to be contained:

```
>>> Color.BLACK in Color.WHITE
False
```

otherwise, if all bits of one flag are in the other flag, True is returned:

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

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

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

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

## 14 How are Enums different?

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

### 14.1 Enum Classes

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

### 14.2 Enum Members (aka instances)

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

While `Enum`, `IntEnum`, `StrEnum`, `Flag`, and `IntFlag` are expected to cover the majority of use-cases, they cannot cover them all. Here are recipes for some different types of enumerations that can be used directly, or as examples for creating one's own.

### 14.3 Omitting values

In many use-cases one doesn't care what the actual value of an enumeration is. There are several ways to define this type of simple enumeration:

- use instances of `auto` for the value
- use instances of `object` as the value
- use a descriptive string as the value
- use a tuple as the value and a custom `__new__()` to replace the tuple with an `int` value

Using any of these methods signifies to the user that these values are not important, and also enables one to add, remove, or reorder members without having to renumber the remaining members.

#### Using `auto`

Using `auto` would look like:

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



```

>>> 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>
>>> 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.

---

## 14.4 OrderedEnum

An ordered enumeration that is not based on `IntEnum` and so maintains the normal Enum invariants (such as not being comparable to other enumerations):

```

>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True

```

## 14.5 DuplicateFreeEnum

Raises an error if a duplicate member name 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:** This is a useful example for subclassing Enum to add or change other behaviors as well as disallowing aliases. If the only desired change is disallowing aliases, the `unique()` decorator can be used instead.

---

## 14.6 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
```

## 14.7 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, Period.day_1]
>>> list(Period)[-2:]
[Period.day_365, Period.day_366]
```

## 14.8 Conforming input to Flag

Creating a Flag enum that is more resilient out-of-bounds results to mathematical operations, you can use the `FlagBoundary.CONFORM` setting:

```
>>> from enum import Flag, CONFORM, auto
>>> class Weekday(Flag, boundary=CONFORM):
...     MONDAY = auto()
...     TUESDAY = auto()
...     WEDNESDAY = auto()
...     THURSDAY = auto()
...     FRIDAY = auto()
...     SATURDAY = auto()
...     SUNDAY = auto()
>>> today = Weekday.TUESDAY
>>> Weekday(today + 22) # what day is three weeks from tomorrow?
>>> Weekday.WEDNESDAY
```

## 15 Subclassing EnumType

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