enum – Enumeration Type

The enum module defines an enumeration type with iteration and comparison capabilities. It can be used to create well-defined symbols for values, instead of using literal integers or strings.

Creating Enumerations

A new enumeration is defined using the `class` syntax by subclassing `Enum` and adding class attributes describing the values.

```
# enum_create.py

import enum


class BugStatus(enum.Enum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1


print('\nMember name: {}'.format(BugStatus.wont_fix.name))
print('Member value: {}'.format(BugStatus.wont_fix.value))
```

The members of the `Enum` are converted to instances as the class is parsed. Each instance has a `name` property corresponding to the member name and a `value` property corresponding to the value assigned to the name in the class definition.

```
$ python3 enum_create.py


Member name: wont_fix
Member value: 4
```

Iteration

Iterating over the enum *class* produces the individual members of the enumeration.

```
# enum_iterate.py

import enum


class BugStatus(enum.Enum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1


for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
```

The members are produced in the order they are declared in the class definition. The names and values are not used to sort them in any way.

them in any way.

```
$ python3 enum_iterate.py

new             = 7
incomplete      = 6
invalid         = 5
wont_fix        = 4
in_progress     = 3
fix_committed   = 2
fix_released    = 1
```

Comparing Enums

Because enumeration members are not ordered, they support only comparison by identity and equality.

```python
# enum_comparison.py

import enum


class BugStatus(enum.Enum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1


actual_state = BugStatus.wont_fix
desired_state = BugStatus.fix_released

print('Equality:',
      actual_state == desired_state,
      actual_state == BugStatus.wont_fix)
print('Identity:',
      actual_state is desired_state,
      actual_state is BugStatus.wont_fix)
print('Ordered by value:')
try:
    print('\n'.join('  ' + s.name for s in sorted(BugStatus)))
except TypeError as err:
    print('  Cannot sort: {}'.format(err))
```

The greater-than and less-than comparison operators raise `TypeError` exceptions.

```
$ python3 enum_comparison.py

Equality: False True
Identity: False True
Ordered by value:
  Cannot sort: '<' not supported between instances of 'BugStatus
' and 'BugStatus'
```

Use the `IntEnum` class for enumerations where the members need to behave more like numbers—for example, to support comparisons.

```python
# enum_intenum.py

import enum


class BugStatus(enum.IntEnum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1


print('Ordered by value:')
print('\n'.join('  ' + s.name for s in sorted(BugStatus)))
```

```
$ python3 enum_intenum.py

Ordered by value:
  fix_released
  fix_committed
  in_progress
  wont_fix
  invalid
  incomplete
  new
```

Unique Enumeration Values

Enum members with the same value are tracked as alias references to the same member object. Aliases do not cause repeated values to be present in the iterator for the Enum.

```
# enum_aliases.py

import enum


class BugStatus(enum.Enum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1

    by_design = 4
    closed = 1


for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))

print('\nSame: by_design is wont_fix: ',
      BugStatus.by_design is BugStatus.wont_fix)
print('Same: closed is fix_released: ',
      BugStatus.closed is BugStatus.fix_released)
```

Because by_design and closed are aliases for other members, they do not appear separately in the output when iterating over the Enum. The canonical name for a member is the first name attached to the value.

```
$ python3 enum_aliases.py

new             = 7
incomplete      = 6
invalid         = 5
wont_fix        = 4
in_progress     = 3
fix_committed   = 2
fix_released    = 1

Same: by_design is wont_fix:  True
Same: closed is fix_released:  True
```

To require all members to have unique values, add the @unique decorator to the Enum.

```
# enum_unique_enforce.py

import enum


@enum.unique
class BugStatus(enum.Enum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1

    # This will trigger an error with unique applied.
    by_design = 4
    closed = 1
```

Members with repeated values trigger a ValueError exception when the Enum class is being interpreted.

```
$ python3 enum_unique_enforce.py

Traceback (most recent call last):
  File "enum_unique_enforce.py", line 11, in <module>
    class BugStatus(enum.Enum):
  File ".../lib/python3.7/enum.py", line 848, in unique
    (enumeration, alias_details))
ValueError: duplicate values found in <enum 'BugStatus'>:
by_design -> wont_fix, closed -> fix_released
```

Creating Enumerations Programmatically

In some cases, it is more convenient to create enumerations programmatically, rather than hard-coding them in a class definition. For those situations, Enum also supports passing the member names and values to the class constructor.

```
# enum_programmatic_create.py

import enum


BugStatus = enum.Enum(
    value='BugStatus',
    names=('fix_released fix_committed in_progress '
           'wont_fix invalid incomplete new'),
)

print('Member: {}'.format(BugStatus.new))

print('\nAll members:')
for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
```

The value argument is the name of the enumeration, which is used to build the representation of members. The names argument lists the members of the enumeration. When a single string is passed, it is split on whitespace and commas, and the resulting tokens are used as names for the members, which are automatically assigned values starting with 1.

```
$ python3 enum_programmatic_create.py

Member: BugStatus.new

All members:
fix_released    = 1
fix_committed   = 2
in_progress     = 3
```

```
wont_fix        = 4
invalid         = 5
incomplete      = 6
new             = 7
```

For more control over the values associated with members, the names string can be replaced with a sequence of two-part tuples or a dictionary mapping names to values.

```
# enum_programmatic_mapping.py

import enum


BugStatus = enum.Enum(
    value='BugStatus',
    names=[
        ('new', 7),
        ('incomplete', 6),
        ('invalid', 5),
        ('wont_fix', 4),
        ('in_progress', 3),
        ('fix_committed', 2),
        ('fix_released', 1),
    ],
)

print('All members:')
for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
```

In this example, a list of two-part tuples is given instead of a single string containing only the member names. This makes it possible to reconstruct the BugStatus enumeration with the members in the same order as the version defined in enum_create.py.

```
$ python3 enum_programmatic_mapping.py

All members:
new             = 7
incomplete      = 6
invalid         = 5
wont_fix        = 4
in_progress     = 3
fix_committed   = 2
fix_released    = 1
```

Non-integer Member Values

Enum member values are not restricted to integers. In fact, any type of object can be associated with a member. If the value is a tuple, the members are passed as individual arguments to __init__().

```
# enum_tuple_values.py

import enum


class BugStatus(enum.Enum):

    new = (7, ['incomplete',
               'invalid',
               'wont_fix',
               'in_progress'])
    incomplete = (6, ['new', 'wont_fix'])
    invalid = (5, ['new'])
    wont_fix = (4, ['new'])
    in_progress = (3, ['new', 'fix_committed'])
    fix_committed = (2, ['in_progress', 'fix_released'])
    fix_released = (1, ['new'])

    def __init__(self, num, transitions):
        self.num = num
        self.transitions = transitions

    def can_transition(self, new_state):
        return new_state.name in self.transitions


print('Name:', BugStatus.in_progress)
print('Value:', BugStatus.in_progress.value)
print('Custom attribute:', BugStatus.in_progress.transitions)
print('Using attribute:',
      BugStatus.in_progress.can_transition(BugStatus.new))
```

In this example, each member value is a tuple containing the numerical ID (such as might be stored in a database) and a list of valid transitions away from the current state.

```
$ python3 enum_tuple_values.py

Name: BugStatus.in_progress
Value: (3, ['new', 'fix_committed'])
Custom attribute: ['new', 'fix_committed']
Using attribute: True
```

For more complex cases, tuples might become unwieldy. Since member values can be any type of object, dictionaries can be used for cases where there are a lot of separate attributes to track for each enum value. Complex values are passed directly to `__init__()` as the only argument other than `self`.

```
# enum_complex_values.py

import enum


class BugStatus(enum.Enum):

    new = {
        'num': 7,
        'transitions': [
            'incomplete',
            'invalid',
            'wont_fix',
            'in_progress',
        ],
    }
    incomplete = {
        'num': 6,
        'transitions': ['new', 'wont_fix'],
    }
    invalid = {
        'num': 5,
        'transitions': ['new'],
    }
    wont_fix = {
        'num': 4,
        'transitions': ['new'],
    }
    in_progress = {
        'num': 3,
        'transitions': ['new', 'fix_committed'],
    }
    fix_committed = {
        'num': 2,
        'transitions': ['in_progress', 'fix_released'],
    }
    fix_released = {
        'num': 1,
        'transitions': ['new'],
    }

    def __init__(self, vals):
        self.num = vals['num']
        self.transitions = vals['transitions']

    def can_transition(self, new_state):
        return new_state.name in self.transitions


print('Name:', BugStatus.in_progress)
print('Value:', BugStatus.in_progress.value)
print('Custom attribute:', BugStatus.in_progress.transitions)
print('Using attribute:',
      BugStatus.in_progress.can_transition(BugStatus.new))
```

This example expresses the same data as the previous example, using dictionaries rather than tuples.

```
$ python3 enum_complex_values.py

Name: BugStatus.in_progress
Value: {'num': 3, 'transitions': ['new', 'fix_committed']}
Custom attribute: ['new', 'fix_committed']
Using attribute: True
```

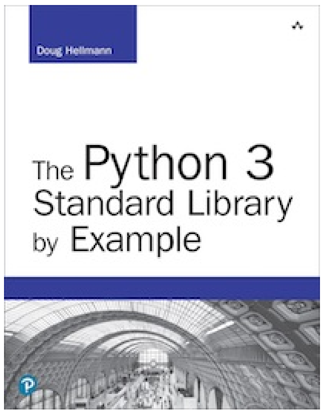
See also

- Standard library documentation for enum
- **PEP 435** – Adding an Enum type to the Python standard library
- flufl.enum – The original inspiration for enum, by Barry Warsaw.

Data Structures   collections — Container Data Types

This page was last updated 2018-12-09.

Navigation
Data Structures
collections — Container Data Types



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?