python / cpython

Code Pull requests 1.4k Actions Security Insights

master

cpython / Lib / enum.py / Jump to

ethanfurman bpo-38250: [Enum] single-bit flags are canonical (GH-24215) … History

24 contributors +12

Raw Blame

1321 lines (1189 sloc) 47.5 KB

```python
import sys
from types import MappingProxyType, DynamicClassAttribute
from builtins import property as _bltin_property, bin as _bltin_bin


__all__ = [
        'EnumMeta',
        'Enum', 'IntEnum', 'StrEnum', 'Flag', 'IntFlag',
        'auto', 'unique',
        'property',
        'FlagBoundary', 'STRICT', 'CONFORM', 'EJECT', 'KEEP',
        ]


# Dummy value for Enum and Flag as there are explicit checks for them
# before they have been created.
# This is also why there are checks in EnumMeta like `if Enum is not None`
Enum = Flag = EJECT = None

def _is_descriptor(obj):
    """
    Returns True if obj is a descriptor, False otherwise.
    """
    return (
            hasattr(obj, '__get__') or
            hasattr(obj, '__set__') or
            hasattr(obj, '__delete__')
            )

def _is_dunder(name):
    """
    Returns True if a __dunder__ name, False otherwise.
    """
    return (
            len(name) > 4 and
            name[:2] == name[-2:] == '__' and
            name[2] != '_' and
            name[-3] != '_'
            )

def _is_sunder(name):
    """
    Returns True if a _sunder_ name, False otherwise.
    """
    return (
            len(name) > 2 and
            name[0] == name[-1] == '_' and
            name[1:2] != '_' and
            name[-2:-1] != '_'
            )

def _is_private(cls_name, name):
    # do not use `re` as `re` imports `enum`
    pattern = '_%s__' % (cls_name, )
    if (
            len(name) >= 5
            and name.startswith(pattern)
            and name[len(pattern)] != '_'
            and (name[-1] != '_' or name[-2] != '_')
        ):
        return True
    else:
        return False

def _is_single_bit(num):
    """
    True if only one bit set in num (should be an int)
    """
    if num == 0:
        return False
    num &= num - 1
    return num == 0

def _make_class_unpicklable(obj):
    """
    Make the given obj un-picklable.

    obj should be either a dictionary, on an Enum
    """
    def _break_on_call_reduce(self, proto):
        raise TypeError('%r cannot be pickled' % self)
    if isinstance(obj, dict):
        obj['__reduce_ex__'] = _break_on_call_reduce
        obj['__module__'] = '<unknown>'
    else:
        setattr(obj, '__reduce_ex__', _break_on_call_reduce)
        setattr(obj, '__module__', '<unknown>')

def _iter_bits_lsb(num):
    while num:
        b = num & (~num + 1)
        yield b
        num ^= b

def bin(num, max_bits=None):
    """
    Like built-in bin(), except negative values are represented in
    twos-compliment, and the leading bit always indicates sign
    (0=positive, 1=negative).

    >>> bin(10)
    '0b0 1010'
    >>> bin(~10)   # ~10 is -11
    '0b1 0101'
    """

    ceiling = 2 ** (num).bit_length()
    if num >= 0:
        s = _bltin_bin(num + ceiling).replace('1', '0', 1)
    else:
        s = _bltin_bin(~num ^ (ceiling - 1) + ceiling)
    sign = s[:3]
    digits = s[3:]
    if max_bits is not None:
        if len(digits) < max_bits:
            digits = (sign[-1] * max_bits + digits)[-max_bits:]
    return "%s %s" % (sign, digits)


_auto_null = object()
class auto:
    """
    Instances are replaced with an appropriate value in Enum class suites.
    """
    value = _auto_null

class property(DynamicClassAttribute):
    """
    This is a descriptor, used to define attributes that act differently
    when accessed through an enum member and through an enum class.
    Instance access is the same as property(), but access to an attribute
    through the enum class will instead look in the class' _member_map_ for
    a corresponding enum member.
    """

    def __get__(self, instance, ownerclass=None):
        if instance is None:
            try:
                return ownerclass._member_map_[self.name]
            except KeyError:
                raise AttributeError(
                        '%s: no attribute %r' % (ownerclass.__name__, self.name)
                        )
        else:
            if self.fget is None:
                raise AttributeError(
                        '%s: no attribute %r' % (ownerclass.__name__, self.name)
                        )
            else:
                return self.fget(instance)

    def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError(
                    "%s: cannot set attribute %r" % (self.clsname, self.name)
                    )
        else:
            return self.fset(instance, value)

    def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError(
                    "%s: cannot delete attribute %r" % (self.clsname, self.name)
                    )
        else:
            return self.fdel(instance)

    def __set_name__(self, ownerclass, name):
        self.name = name
        self.clsname = ownerclass.__name__
```

```python


class _proto_member:
    """
    intermediate step for enum members between class execution and final creation
    """

    def __init__(self, value):
        self.value = value

    def __set_name__(self, enum_class, member_name):
        """
        convert each quasi-member into an instance of the new enum class
        """
        # first step: remove ourself from enum_class
        delattr(enum_class, member_name)
        # second step: create member based on enum_class
        value = self.value
        if not isinstance(value, tuple):
            args = (value, )
        else:
            args = value
        if enum_class._member_type_ is tuple:   # special case for tuple enums
            args = (args, )     # wrap it one more time
        if not enum_class._use_args_:
            enum_member = enum_class._new_member_(enum_class)
            if not hasattr(enum_member, '_value_'):
                enum_member._value_ = value
        else:
            enum_member = enum_class._new_member_(enum_class, *args)
            if not hasattr(enum_member, '_value_'):
                if enum_class._member_type_ is object:
                    enum_member._value_ = value
                else:
                    try:
                        enum_member._value_ = enum_class._member_type_(*args)
                    except Exception as exc:
                        raise TypeError(
                                '_value_ not set in __new__, unable to create it'
                                ) from None
        value = enum_member._value_
        enum_member._name_ = member_name
        enum_member.__objclass__ = enum_class
        enum_member.__init__(*args)
        enum_member._sort_order_ = len(enum_class._member_names_)
        # If another member with the same value was already defined, the
        # new member becomes an alias to the existing one.
        for name, canonical_member in enum_class._member_map_.items():
            if canonical_member._value_ == enum_member._value_:
                enum_member = canonical_member
                break
        else:
            # this could still be an alias if the value is multi-bit and the
            # class is a flag class
            if (
                    Flag is None
                    or not issubclass(enum_class, Flag)
                ):
                # no other instances found, record this member in _member_names_
                enum_class._member_names_.append(member_name)
            elif (
                    Flag is not None
                    and issubclass(enum_class, Flag)
                    and _is_single_bit(value)
                ):
                # no other instances found, record this member in _member_names_
                enum_class._member_names_.append(member_name)
        # get redirect in place before adding to _member_map_
        # but check for other instances in parent classes first
        need_override = False
        descriptor = None
        for base in enum_class.__mro__[1:]:
            descriptor = base.__dict__.get(member_name)
            if descriptor is not None:
                if isinstance(descriptor, (property, DynamicClassAttribute)):
                    break
                else:
                    need_override = True
                    # keep looking for an enum.property
        if descriptor and not need_override:
            # previous enum.property found, no further action needed
            pass
        else:
            redirect = property()
            redirect.__set_name__(enum_class, member_name)
            if descriptor and need_override:
                # previous enum.property found, but some other inherited attribute
                # is in the way; copy fget, fset, fdel to this one
                redirect.fget = descriptor.fget
                redirect.fset = descriptor.fset
                redirect.fdel = descriptor.fdel
            setattr(enum_class, member_name, redirect)
        # now add to _member_map_ (even aliases)
        enum_class._member_map_[member_name] = enum_member
        try:
            # This may fail if value is not hashable. We can't add the value
                # to the map, and by-value lookups for this value will be
                # linear.
                enum_class._value2member_map_.setdefault(value, enum_member)
            except TypeError:
                pass


class _EnumDict(dict):
    """
    Track enum member order and ensure member names are not reused.

    EnumMeta will use the names found in self._member_names as the
    enumeration member names.
    """
    def __init__(self):
        super().__init__()
        self._member_names = []
        self._last_values = []
        self._ignore = []
        self._auto_called = False

    def __setitem__(self, key, value):
        """
        Changes anything not dundered or not a descriptor.

        If an enum member name is used twice, an error is raised; duplicate
        values are not checked for.

        Single underscore (sunder) names are reserved.
        """
        if _is_private(self._cls_name, key):
            # do nothing, name will be a normal attribute
            pass
        elif _is_sunder(key):
            if key not in (
                    '_order_', '_create_pseudo_member_',
                    '_generate_next_value_', '_missing_', '_ignore_',
                    '_iter_member_', '_iter_member_by_value_', '_iter_member_by_def_',
                    ):
                raise ValueError(
                        '_sunder_ names, such as %r, are reserved for future Enum use'
                        % (key, )
                        )
            if key == '_generate_next_value_':
                # check if members already defined as auto()
                if self._auto_called:
                    raise TypeError("_generate_next_value_ must be defined before members")
                setattr(self, '_generate_next_value', value)
            elif key == '_ignore_':
                if isinstance(value, str):
                    value = value.replace(',',' ').split()
                else:
                    value = list(value)
                self._ignore = value
                already = set(value) & set(self._member_names)
                if already:
                    raise ValueError(
                            '_ignore_ cannot specify already set names: %r'
                            % (already, )
                            )
        elif _is_dunder(key):
            if key == '__order__':
                key = '_order_'
        elif key in self._member_names:
            # descriptor overwriting an enum?
            raise TypeError('%r already defined as: %r' % (key, self[key]))
        elif key in self._ignore:
            pass
        elif not _is_descriptor(value):
            if key in self:
                # enum overwriting a descriptor?
                raise TypeError('%r already defined as: %r' % (key, self[key]))
            if isinstance(value, auto):
                if value.value == _auto_null:
                    value.value = self._generate_next_value(
                            key, 1, len(self._member_names), self._last_values[:],
                            )
                    self._auto_called = True
                value = value.value
            self._member_names.append(key)
            self._last_values.append(value)
        super().__setitem__(key, value)

    def update(self, members, **more_members):
        try:
            for name in members.keys():
                self[name] = members[name]
        except AttributeError:
            for name, value in members:
                self[name] = value
        for name, value in more_members.items():
            self[name] = value


class EnumMeta(type):
    """
    Metaclass for Enum
    """

    @classmethod
    def __prepare__(metacls, cls, bases, **kwds):
        # check that previous enum members do not exist
        metacls._check_for_existing_members(cls, bases)
        # create the namespace dict
        enum_dict = _EnumDict()
        enum_dict._cls_name = cls
        # inherit previous flags and _generate_next_value_ function
        member_type, first_enum = metacls._get_mixins_(cls, bases)
        if first_enum is not None:
            enum_dict['_generate_next_value_'] = getattr(
                    first_enum, '_generate_next_value_', None,
                    )
        return enum_dict

    def __new__(metacls, cls, bases, classdict, boundary=None, **kwds):
        # an Enum class is final once enumeration items have been defined; it
        # cannot be mixed with other types (int, float, etc.) if it has an
        # inherited __new__ unless a new __new__ is defined (or the resulting
        # class will fail).
        #
        # remove any keys listed in _ignore_
        classdict.setdefault('_ignore_', []).append('_ignore_')
        ignore = classdict['_ignore_']
        for key in ignore:
            classdict.pop(key, None)
        #
        # grab member names
        member_names = classdict._member_names
        #
        # check for illegal enum names (any others?)
        invalid_names = set(member_names) & {'mro', ''}
        if invalid_names:
            raise ValueError('Invalid enum member name: {0}'.format(
                ','.join(invalid_names)))
        #
        # adjust the sunders
        _order_ = classdict.pop('_order_', None)
        # convert to normal dict
        classdict = dict(classdict.items())
        #
        # data type of member and the controlling Enum class
        member_type, first_enum = metacls._get_mixins_(cls, bases)
        __new__, save_new, use_args = metacls._find_new_(
                classdict, member_type, first_enum,
                )
        classdict['_new_member_'] = __new__
        classdict['_use_args_'] = use_args
        #
        # convert future enum members into temporary _proto_members
        # and record integer values in case this will be a Flag
        flag_mask = 0
        for name in member_names:
            value = classdict[name]
            if isinstance(value, int):
                flag_mask |= value
            classdict[name] = _proto_member(value)
        #
        # house-keeping structures
        classdict['_member_names_'] = []
        classdict['_member_map_'] = {}
        classdict['_value2member_map_'] = {}
        classdict['_member_type_'] = member_type
        #
        # Flag structures (will be removed if final class is not a Flag
        classdict['_boundary_'] = (
                boundary
                or getattr(first_enum, '_boundary_', None)
                )
        classdict['_flag_mask_'] = flag_mask
        classdict['_all_bits_'] = 2 ** ((flag_mask).bit_length()) - 1
        classdict['_inverted_'] = None
        #
        # If a custom type is mixed into the Enum, and it does not know how
        # to pickle itself, pickle.dumps will succeed but pickle.loads will
        # fail.  Rather than have the error show up later and possibly far
        # from the source, sabotage the pickle protocol for this class so
        # that pickle.dumps also fails.
        #
        # However, if the new class implements its own __reduce_ex__, do not
        # sabotage -- it's on them to make sure it works correctly.  We use
        # __reduce_ex__ instead of any of the others as it is preferred by
        # pickle over __reduce__, and it handles all pickle protocols.
        if '__reduce_ex__' not in classdict:
            if member_type is not object:
                methods = ('__getnewargs_ex__', '__getnewargs__',
                        '__reduce_ex__', '__reduce__')
                if not any(m in member_type.__dict__ for m in methods):
                    _make_class_unpicklable(classdict)
        #
        # create a default docstring if one has not been provided
        if '__doc__' not in classdict:
            classdict['__doc__'] = 'An enumeration.'
        try:
            exc = None
            enum_class = super().__new__(metacls, cls, bases, classdict, **kwds)
        except RuntimeError as e:
            # any exceptions raised by member.__new__ will get converted to a
            # RuntimeError, so get that original exception back and raise it instead
            exc = e.__cause__ or e
        if exc is not None:
            raise exc
        #
        # double check that repr and friends are not the mixin's or various
        # things break (such as pickle)
        # however, if the method is defined in the Enum itself, don't replace
        # it
        for name in ('__repr__', '__str__', '__format__', '__reduce_ex__'):
            if name in classdict:
                continue
            class_method = getattr(enum_class, name)
            obj_method = getattr(member_type, name, None)
            enum_method = getattr(first_enum, name, None)
            if obj_method is not None and obj_method is class_method:
                setattr(enum_class, name, enum_method)
        #
        # replace any other __new__ with our own (as long as Enum is not None,
        # anyway) -- again, this is to support pickle
        if Enum is not None:
            # if the user defined their own __new__, save it before it gets
            # clobbered in case they subclass later
            if save_new:
                enum_class.__new_member__ = __new__
            enum_class.__new__ = Enum.__new__
        #
        # py3 support for definition order (helps keep py2/py3 code in sync)
        #
        # _order_ checking is spread out into three/four steps
        # - if enum_class is a Flag:
        #   - remove any non-single-bit flags from _order_
        # - remove any aliases from _order_
        # - check that _order_ and _member_names_ match
        #
        # step 1: ensure we have a list
        if _order_ is not None:
            if isinstance(_order_, str):
                _order_ = _order_.replace(',', ' ').split()
        #
        # remove Flag structures if final class is not a Flag
        if (
                Flag is None and cls != 'Flag'
```

```python
                    or Flag is not None and not issubclass(enum_class, Flag)
                ):
                delattr(enum_class, '_boundary_')
                delattr(enum_class, '_flag_mask_')
                delattr(enum_class, '_all_bits_')
                delattr(enum_class, '_inverted_')
            elif Flag is not None and issubclass(enum_class, Flag):
                # ensure _all_bits_ is correct and there are no missing flags
                single_bit_total = 0
                multi_bit_total = 0
                for flag in enum_class._member_map_.values():
                    flag_value = flag._value_
                    if _is_single_bit(flag_value):
                        single_bit_total |= flag_value
                    else:
                        # multi-bit flags are considered aliases
                        multi_bit_total |= flag_value
                if enum_class._boundary_ is not KEEP:
                    missed = list(_iter_bits_lsb(multi_bit_total & ~single_bit_total))
                    if missed:
                        raise TypeError(
                                'invalid Flag %r -- missing values: %s'
                                % (cls, ', '.join((str(i) for i in missed)))
                                )
                enum_class._flag_mask_ = single_bit_total
                #
                # set correct __iter__
                member_list = [m._value_ for m in enum_class]
                if member_list != sorted(member_list):
                    enum_class._iter_member_ = enum_class._iter_member_by_def_
                if _order_:
                    # _order_ step 2: remove any items from _order_ that are not single-bit
                    _order_ = [
                            o
                            for o in _order_
                            if o not in enum_class._member_map_ or _is_single_bit(enum_class[o]._va
                            ]
            #
            if _order_:
                # _order_ step 3: remove aliases from _order_
                _order_ = [
                        o
                        for o in _order_
                        if (
                            o not in enum_class._member_map_
                            or
                            (o in enum_class._member_map_ and o in enum_class._member_names_)
                            )]
                # _order_ step 4: verify that _order_ and _member_names_ match
                if _order_ != enum_class._member_names_:
                    raise TypeError(
                            'member order does not match _order_:\n%r\n%r'
                            % (enum_class._member_names_, _order_)
                            )
            #
            return enum_class

        def __bool__(self):
            """
            classes/types should always be True.
            """
            return True

        def __call__(cls, value, names=None, *, module=None, qualname=None, type=None, start=1, bou
            """
            Either returns an existing member, or creates a new enum class.

            This method is used both when an enum class is given a value to match
            to an enumeration member (i.e. Color(3)) and for the functional API
            (i.e. Color = Enum('Color', names='RED GREEN BLUE')).

            When used for the functional API:

            `value` will be the name of the new class.

            `names` should be either a string of white-space/comma delimited names
            (values will start at `start`), or an iterator/mapping of name, value pairs.

            `module` should be set to the module this class is being created in;
            if it is not set, an attempt to find that module will be made, but if
            it fails the class will not be picklable.

            `qualname` should be set to the actual location this class can be found
            at in its module; by default it is set to the global scope.  If this is
            not correct, unpickling will fail in some circumstances.

            `type`, if set, will be mixed in as the first base class.
            """
            if names is None:  # simple value lookup
                return cls.__new__(cls, value)
            # otherwise, functional API: we're creating a new Enum type
            return cls._create_(
                    value,
                    names,
                    module=module,
                    qualname=qualname,
                    type=type,
                    start=start,
                    boundary=boundary,
                    )

        def __contains__(cls, member):
            if not isinstance(member, Enum):
                raise TypeError(
                    "unsupported operand type(s) for 'in': '%s' and '%s'" % (
                        type(member).__qualname__, cls.__class__.__qualname__))
            return isinstance(member, cls) and member._name_ in cls._member_map_

        def __delattr__(cls, attr):
            # nicer error message when someone tries to delete an attribute
            # (see issue19025).
            if attr in cls._member_map_:
                raise AttributeError("%s: cannot delete Enum member %r." % (cls.__name__, attr))
            super().__delattr__(attr)

        def __dir__(self):
            return (
                    ['__class__', '__doc__', '__members__', '__module__']
                    + self._member_names_
                    )

        def __getattr__(cls, name):
            """
            Return the enum member matching `name`

            We use __getattr__ instead of descriptors or inserting into the enum
            class' __dict__ in order to support `name` and `value` being both
            properties for enum members (which live in the class' __dict__) and
            enum members themselves.
            """
            if _is_dunder(name):
                raise AttributeError(name)
            try:
                return cls._member_map_[name]
            except KeyError:
                raise AttributeError(name) from None

        def __getitem__(cls, name):
            return cls._member_map_[name]

        def __iter__(cls):
            """
            Returns members in definition order.
            """
```

```python
            return (cls._member_map_[name] for name in cls._member_names_)

    def __len__(cls):
        return len(cls._member_names_)

    @_bltin_property
    def __members__(cls):
        """
        Returns a mapping of member name->value.

        This mapping lists all enum members, including aliases. Note that this
        is a read-only view of the internal mapping.
        """
        return MappingProxyType(cls._member_map_)

    def __repr__(cls):
        return "<enum %r>" % cls.__name__

    def __reversed__(cls):
        """
        Returns members in reverse definition order.
        """
        return (cls._member_map_[name] for name in reversed(cls._member_names_))

    def __setattr__(cls, name, value):
        """
        Block attempts to reassign Enum members.

        A simple assignment to the class namespace only changes one of the
        several possible ways to get an Enum member from the Enum class,
        resulting in an inconsistent Enumeration.
        """
        member_map = cls.__dict__.get('_member_map_', {})
        if name in member_map:
            raise AttributeError('Cannot reassign members.')
        super().__setattr__(name, value)

    def _create_(cls, class_name, names, *, module=None, qualname=None, type=None, start=1, bou
        """
        Convenience method to create a new Enum class.

        `names` can be:

        * A string containing member names, separated either with spaces or
          commas.  Values are incremented by 1 from `start`.
        * An iterable of member names.  Values are incremented by 1 from `start`.
        * An iterable of (member name, value) pairs.
        * A mapping of member name -> value pairs.
        """
        metacls = cls.__class__
        bases = (cls, ) if type is None else (type, cls)
        _, first_enum = cls._get_mixins_(cls, bases)
        classdict = metacls.__prepare__(class_name, bases)

        # special processing needed for names?
        if isinstance(names, str):
            names = names.replace(',', ' ').split()
        if isinstance(names, (tuple, list)) and names and isinstance(names[0], str):
            original_names, names = names, []
            last_values = []
            for count, name in enumerate(original_names):
                value = first_enum._generate_next_value_(name, start, count, last_values[:])
                last_values.append(value)
                names.append((name, value))

        # Here, names is either an iterable of (name, value) or a mapping.
        for item in names:
            if isinstance(item, str):
                member_name, member_value = item, names[item]
            else:
                member_name, member_value = item
            classdict[member_name] = member_value

        # TODO: replace the frame hack if a blessed way to know the calling
        # module is ever developed
        if module is None:
            try:
                module = sys._getframe(2).f_globals['__name__']
            except (AttributeError, ValueError, KeyError):
                pass
        if module is None:
            _make_class_unpicklable(classdict)
        else:
            classdict['__module__'] = module
        if qualname is not None:
            classdict['__qualname__'] = qualname

        return metacls.__new__(metacls, class_name, bases, classdict, boundary=boundary)

    def _convert_(cls, name, module, filter, source=None, boundary=None):
        """
        Create a new Enum subclass that replaces a collection of global constants
        """
        # convert all constants from source (or module) that pass filter() to
        # a new Enum called name, and export the enum and its members back to
        # module;
        # also, replace the __reduce_ex__ method so unpickling works in
        # previous Python versions
        module_globals = vars(sys.modules[module])
        if source:
            source = vars(source)
        else:
            source = module_globals
        # _value2member_map_ is populated in the same order every time
        # for a consistent reverse mapping of number to name when there
        # are multiple names for the same number.
        members = [
                (name, value)
                for name, value in source.items()
                if filter(name)]
        try:
            # sort by value
            members.sort(key=lambda t: (t[1], t[0]))
        except TypeError:
            # unless some values aren't comparable, in which case sort by name
            members.sort(key=lambda t: t[0])
        cls = cls(name, members, module=module, boundary=boundary or KEEP)
        cls.__reduce_ex__ = _reduce_ex_by_name
        module_globals.update(cls.__members__)
        module_globals[name] = cls
        return cls

    @staticmethod
    def _check_for_existing_members(class_name, bases):
        for chain in bases:
            for base in chain.__mro__:
                if issubclass(base, Enum) and base._member_names_:
                    raise TypeError(
                            "%s: cannot extend enumeration %r"
                            % (class_name, base.__name__)
                            )

    @staticmethod
    def _get_mixins_(class_name, bases):
        """
        Returns the type for creating enum members, and the first inherited
        enum class.

        bases: the tuple of bases that was given to __new__
        """
        if not bases:
            return object, Enum

        def _find_data_type(bases):
```

```python
            data_types = []
            for chain in bases:
                candidate = None
                for base in chain.__mro__:
                    if base is object:
                        continue
                    elif issubclass(base, Enum):
                        if base._member_type_ is not object:
                            data_types.append(base._member_type_)
                            break
                    elif '__new__' in base.__dict__:
                        if issubclass(base, Enum):
                            continue
                        data_types.append(candidate or base)
                        break
                    else:
                        candidate = base
            if len(data_types) > 1:
                raise TypeError('%r: too many data types: %r' % (class_name, data_types))
            elif data_types:
                return data_types[0]
            else:
                return None

        # ensure final parent class is an Enum derivative, find any concrete
        # data type, and check that Enum has no members
        first_enum = bases[-1]
        if not issubclass(first_enum, Enum):
            raise TypeError("new enumerations should be created as "
                    "`EnumName([mixin_type, ...] [data_type,] enum_type)`")
        member_type = _find_data_type(bases) or object
        if first_enum._member_names_:
            raise TypeError("Cannot extend enumerations")
        return member_type, first_enum

    @staticmethod
    def _find_new_(classdict, member_type, first_enum):
        """
        Returns the __new__ to be used for creating the enum members.

        classdict: the class dictionary given to __new__
        member_type: the data type whose __new__ will be used by default
        first_enum: enumeration to check for an overriding __new__
        """
        # now find the correct __new__, checking to see of one was defined
        # by the user; also check earlier enum classes in case a __new__ was
        # saved as __new_member__
        __new__ = classdict.get('__new__', None)

        # should __new__ be saved as __new_member__ later?
        save_new = __new__ is not None

        if __new__ is None:
            # check all possibles for __new_member__ before falling back to
            # __new__
            for method in ('__new_member__', '__new__'):
                for possible in (member_type, first_enum):
                    target = getattr(possible, method, None)
                    if target not in {
                            None,
                            None.__new__,
                            object.__new__,
                            Enum.__new__,
                            }:
                        __new__ = target
                        break
                if __new__ is not None:
                    break
            else:
                __new__ = object.__new__

        # if a non-object.__new__ is used then whatever value/tuple was
        # assigned to the enum member name will be passed to __new__ and to the
        # new enum member's __init__
        if __new__ is object.__new__:
            use_args = False
        else:
            use_args = True
        return __new__, save_new, use_args


class Enum(metaclass=EnumMeta):
    """
    Generic enumeration.

    Derive from this class to define new enumerations.
    """

    def __new__(cls, value):
        # all enum instances are actually created during class construction
        # without calling this method; this method is called by the metaclass'
        # __call__ (i.e. Color(3) ), and by pickle
        if type(value) is cls:
            # For lookups like Color(Color.RED)
            return value
        # by-value search for a matching enum member
        # see if it's in the reverse mapping (for hashable values)
        try:
            return cls._value2member_map_[value]
        except KeyError:
            # Not found, no need to do long O(n) search
            pass
        except TypeError:
            # not there, now do long search -- O(n) behavior
            for member in cls._member_map_.values():
                if member._value_ == value:
                    return member
        # still not found -- try _missing_ hook
        try:
            exc = None
            result = cls._missing_(value)
        except Exception as e:
            exc = e
            result = None
        if isinstance(result, cls):
            return result
        elif (
                Flag is not None and issubclass(cls, Flag)
                and cls._boundary_ is EJECT and isinstance(result, int)
            ):
            return result
        else:
            ve_exc = ValueError("%r is not a valid %s" % (value, cls.__qualname__))
            if result is None and exc is None:
                raise ve_exc
            elif exc is None:
                exc = TypeError(
                        'error in %s._missing_: returned %r instead of None or a valid member'
                        % (cls.__name__, result)
                        )
            if not isinstance(exc, ValueError):
                exc.__context__ = ve_exc
            raise exc

    def _generate_next_value_(name, start, count, last_values):
        """
        Generate the next value when not given.

        name: the name of the member
        start: the initial start value or None
        count: the number of existing members
        last_value: the last value assigned or None
        """
        for last_value in reversed(last_values):
            try:
                return last_value + 1
            except TypeError:
                pass
        else:
            return start

    @classmethod
    def _missing_(cls, value):
        return None

    def __repr__(self):
        return "<%s.%s: %r>" % (
                self.__class__.__name__, self._name_, self._value_)

    def __str__(self):
        return "%s.%s" % (self.__class__.__name__, self._name_)

    def __dir__(self):
        """
        Returns all members and all public methods
        """
        added_behavior = [
                m
                for cls in self.__class__.mro()
                for m in cls.__dict__
                if m[0] != '_' and m not in self._member_map_
                ] + [m for m in self.__dict__ if m[0] != '_']
        return (['__class__', '__doc__', '__module__'] + added_behavior)

    def __format__(self, format_spec):
        """
        Returns format using actual value type unless __str__ has been overridden.
        """
        # mixed-in Enums should use the mixed-in type's __format__, otherwise
        # we can get strange results with the Enum name showing up instead of
        # the value

        # pure Enum branch, or branch with __str__ explicitly overridden
        str_overridden = type(self).__str__ not in (Enum.__str__, Flag.__str__)
        if self._member_type_ is object or str_overridden:
            cls = str
            val = str(self)
        # mix-in branch
        else:
            cls = self._member_type_
            val = self._value_
        return cls.__format__(val, format_spec)
```

```python

    def __hash__(self):
        return hash(self._name_)

    def __reduce_ex__(self, proto):
        return self.__class__, (self._value_, )

    # enum.property is used to provide access to the `name` and
    # `value` attributes of enum members while keeping some measure of
    # protection from modification, while still allowing for an enumeration
    # to have members named `name` and `value`.  This works because enumeration
    # members are not set directly on the enum class; they are kept in a
    # separate structure, _member_map_, which is where enum.property looks for
    # them

    @property
    def name(self):
        """The name of the Enum member."""
        return self._name_

    @property
    def value(self):
        """The value of the Enum member."""
        return self._value_


class IntEnum(int, Enum):
    """
    Enum where members are also (and must be) ints
    """


class StrEnum(str, Enum):
    """
    Enum where members are also (and must be) strings
    """

    def __new__(cls, *values):
        if len(values) > 3:
            raise TypeError('too many arguments for str(): %r' % (values, ))
        if len(values) == 1:
            # it must be a string
            if not isinstance(values[0], str):
                raise TypeError('%r is not a string' % (values[0], ))
        if len(values) >= 2:
            # check that encoding argument is a string
            if not isinstance(values[1], str):
                raise TypeError('encoding must be a string, not %r' % (values[1], ))
        if len(values) == 3:
            # check that errors argument is a string
            if not isinstance(values[2], str):
                raise TypeError('errors must be a string, not %r' % (values[2]))
        value = str(*values)
        member = str.__new__(cls, value)
        member._value_ = value
        return member

    __str__ = str.__str__

    def _generate_next_value_(name, start, count, last_values):
        """
        Return the lower-cased version of the member name.
        """
        return name.lower()


def _reduce_ex_by_name(self, proto):
    return self.name

class FlagBoundary(StrEnum):
    """
    control how out of range values are handled
    "strict" -> error is raised  [default for Flag]
    "conform" -> extra bits are discarded
    "eject" -> lose flag status [default for IntFlag]
    "keep" -> keep flag status and all bits
    """
    STRICT = auto()
    CONFORM = auto()
    EJECT = auto()
    KEEP = auto()
STRICT, CONFORM, EJECT, KEEP = FlagBoundary


class Flag(Enum, boundary=STRICT):
    """
    Support for flags
    """

    def _generate_next_value_(name, start, count, last_values):
        """
        Generate the next value when not given.

        name: the name of the member
        start: the initial start value or None
        count: the number of existing members
        last_value: the last value assigned or None
        """
        if not count:
            return start if start is not None else 1
        last_value = max(last_values)
        try:
            high_bit = _high_bit(last_value)
        except Exception:
            raise TypeError('Invalid Flag value: %r' % last_value) from None
        return 2 ** (high_bit+1)

    @classmethod
    def _iter_member_by_value_(cls, value):
        """
        Extract all members from the value in definition (i.e. increasing value) order.
        """
        for val in _iter_bits_lsb(value & cls._flag_mask_):
            yield cls._value2member_map_.get(val)

    _iter_member_ = _iter_member_by_value_

    @classmethod
    def _iter_member_by_def_(cls, value):
        """
        Extract all members from the value in definition order.
        """
        yield from sorted(
                cls._iter_member_by_value_(value),
                key=lambda m: m._sort_order_,
                )

    @classmethod
    def _missing_(cls, value):
        """
        Create a composite member iff value contains only members.
        """
        if not isinstance(value, int):
            raise ValueError(
                    "%r is not a valid %s" % (value, cls.__qualname__)
                    )
        # check boundaries
        # - value must be in range (e.g. -16 <-> +15, i.e. ~15 <-> 15)
        # - value must not include any skipped flags (e.g. if bit 2 is not
        #   defined, then 0d10 is invalid)
        flag_mask = cls._flag_mask_
        all_bits = cls._all_bits_
        neg_value = None
        if (
                not ~all_bits <= value <= all_bits
                or value & (all_bits ^ flag_mask)
            ):
            if cls._boundary_ is STRICT:
                max_bits = max(value.bit_length(), flag_mask.bit_length())
                raise ValueError(
                        "%s: invalid value: %r\n    given %s\n  allowed %s" % (
                            cls.__name__, value, bin(value, max_bits), bin(flag_mask, max_bits)
                            ))
            elif cls._boundary_ is CONFORM:
                value = value & flag_mask
            elif cls._boundary_ is EJECT:
                return value
            elif cls._boundary_ is KEEP:
                if value < 0:
                    value = (
                            max(all_bits+1, 2**(value.bit_length()))
                            + value
                            )
            else:
                raise ValueError(
                        'unknown flag boundary: %r' % (cls._boundary_, )
                        )
        if value < 0:
            neg_value = value
            value = all_bits + 1 + value
        # get members and unknown
        unknown = value & ~flag_mask
        member_value = value & flag_mask
        if unknown and cls._boundary_ is not KEEP:
            raise ValueError(
                    '%s(%r) -->  unknown values %r [%s]'
                    % (cls.__name__, value, unknown, bin(unknown))
                    )
        # normal Flag?
        __new__ = getattr(cls, '__new_member__', None)
        if cls._member_type_ is object and not __new__:
            # construct a singleton enum pseudo-member
            pseudo_member = object.__new__(cls)
        else:
            pseudo_member = (__new__ or cls._member_type_.__new__)(cls, value)
        if not hasattr(pseudo_member, 'value'):
            pseudo_member._value_ = value
        if member_value:
            pseudo_member._name_ = '|'.join([
                m._name_ for m in cls._iter_member_(member_value)
                ])
            if unknown:
                pseudo_member._name_ += '|0x%x' % unknown
        else:
            pseudo_member._name_ = None
        # use setdefault in case another thread already created a composite
        # with this value, but only if all members are known
        # note: zero is a special case -- add it
        if not unknown:
            pseudo_member = cls._value2member_map_.setdefault(value, pseudo_member)
            if neg_value is not None:
                cls._value2member_map_[neg_value] = pseudo_member
        return pseudo_member

    def __contains__(self, other):
        """
        Returns True if self has at least the same flags set as other.
        """
        if not isinstance(other, self.__class__):
            raise TypeError(
                "unsupported operand type(s) for 'in': '%s' and '%s'" % (
                    type(other).__qualname__, self.__class__.__qualname__))
        if other._value_ == 0 or self._value_ == 0:
            return False
        return other._value_ & self._value_ == other._value_

    def __iter__(self):
        """
        Returns flags in definition order.
        """
        yield from self._iter_member_(self._value_)

    def __len__(self):
        return self._value_.bit_count()

    def __repr__(self):
        cls = self.__class__
        if self._name_ is not None:
            return '<%s.%s: %r>' % (cls.__name__, self._name_, self._value_)
        else:
            # only zero is unnamed by default
            return '<%s: %r>' % (cls.__name__, self._value_)

    def __str__(self):
        cls = self.__class__
        if self._name_ is not None:
            return '%s.%s' % (cls.__name__, self._name_)
        else:
            return '%s(%s)' % (cls.__name__, self._value_)

```

```python
    def __bool__(self):
        return bool(self._value_)

    def __or__(self, other):
        if not isinstance(other, self.__class__):
            return NotImplemented
        return self.__class__(self._value_ | other._value_)

    def __and__(self, other):
        if not isinstance(other, self.__class__):
            return NotImplemented
        return self.__class__(self._value_ & other._value_)

    def __xor__(self, other):
        if not isinstance(other, self.__class__):
            return NotImplemented
        return self.__class__(self._value_ ^ other._value_)

    def __invert__(self):
        if self._inverted_ is None:
            if self._boundary_ is KEEP:
                # use all bits
                self._inverted_ = self.__class__(~self._value_)
            else:
                # calculate flags not in this member
                self._inverted_ = self.__class__(self._flag_mask_ ^ self._value_)
            self._inverted_._inverted_ = self
        return self._inverted_


class IntFlag(int, Flag, boundary=EJECT):
    """
    Support for integer-based Flags
    """

    def __or__(self, other):
        if isinstance(other, self.__class__):
            other = other._value_
        elif isinstance(other, int):
            other = other
        else:
            return NotImplemented
        value = self._value_
        return self.__class__(value | other)

    def __and__(self, other):
        if isinstance(other, self.__class__):
            other = other._value_
        elif isinstance(other, int):
            other = other
        else:
            return NotImplemented
        value = self._value_
        return self.__class__(value & other)

    def __xor__(self, other):
        if isinstance(other, self.__class__):
            other = other._value_
        elif isinstance(other, int):
            other = other
        else:
            return NotImplemented
        value = self._value_
        return self.__class__(value ^ other)

    __ror__ = __or__
    __rand__ = __and__
    __rxor__ = __xor__
    __invert__ = Flag.__invert__

def _high_bit(value):
    """
    returns index of highest bit, or -1 if value is zero or negative
    """
    return value.bit_length() - 1

def unique(enumeration):
    """
    Class decorator for enumerations ensuring unique member values.
    """
    duplicates = []
    for name, member in enumeration.__members__.items():
        if name != member.name:
            duplicates.append((name, member.name))
    if duplicates:
        alias_details = ', '.join(
                ["%s -> %s" % (alias, name) for (alias, name) in duplicates])
        raise ValueError('duplicate values found in %r: %s' %
                (enumeration, alias_details))
    return enumeration

def _power_of_two(value):
    if value < 1:
        return False
    return value == 2 ** _high_bit(value)
```