greenland-base

Release 0.0.6

M E Leypold

Oct 04, 2024

CONTENTS:

CHAPTER

ONE

MODULE *GREENLAND.BASE.ENUMS*

Provides an enumeration (Enum) type as python objects.

The way programming languages handle "enums" — a small set of discrete objects belonging together, like primary colors or days of the week – is often influenced by the capabilities of the programming language whithin which such a facility exists. Very often the emphasis is on providing names (identifiers) for integers as if the central semantics of e.g. a weekday is it being a number. Enums in C certainly work like this.

One unfortunate consequence of these types of design decisions is, that it is difficult to express in type annotations that a certain parameter should only take members of a specific enum. The conclusion we must draw is, that it is desirable that enum types correspond to classes and that the enum members are instances of those classes, so it is possible to express the following:

```
def foo(day: Weekday):
    ...
```

The built-in enums of *Python* work like this already. The author of this package also became (admittedly) aware of the built-in enums too late, so went down a slightly different (but possibly simpler) path.

This package provides enums as a type where the enumeration type is a class (almost like any other) which only has a finite, explicitely specified number of instances (the members) and where the enum member can be enriched with almost any kind of additional behaviour.

1.1 Constructing enum types with `Enum`

class **Enum**(*name_or_ord: str | int*, *ord: int | None = None*)

The super class of all enum types (= classes).

Enum types are created by deriving from *Enum*. Members are added by calling the constructor and assigning to an identifier in the same namespace as the class definition before calling `finalize()`.

After `finalize()`, the constructor will just either return a previously existing member of the enum type or raise a `ParseError`.

Parameters

- **name_or_ord** (*str | int*) – Before finalizing the enum type only the name of the enum member — a `str` – is allowed here. After finalization both a `str` and a `int` here are allowed and do not construct a new member, but rather retrieve a member of the give *name* or *ord* respectively. A `ParseError` will be raised if such a member does not exist.
- **ord** (*int | None*) – Before finalizing the the enum type an optional `ord` integer for the member. After finalization this argument must not be given.

Listing 1: Define enum type and members:

```
from greenland.base.enums import Enum


class Direction(Enum):
    pass


NORTH  = Direction('NORTH')
EAST = Direction('EAST')
SOUTH = Direction('SOUTH')
WEST = Direction('WEST')

Direction.finalize()
```

`finalize()` will actually do two things:

- Lock the type so that further calls to the constructor result either in retrieval of an already defined member or in raising a `ParseError` if no such member exists.
- Check if all members have been bound to names in the same namespace where the enum class has been defined.

Listing 2: Behavior after finalization:

```
assert Direction('SOUTH') == SOUTH

with pytest.raises(AssertionError):
    _ = Direction('FOO')
```

Typically enums are global types. I cannot currently see (except for testing purposes) much of an application for defining an enum in a local namespace.

Regardless, if one desires to do so, this is possible, but `finalize()` needs to be called with `locals()`:

Listing 3: Local enum definition:

```
class Turn(Enum):
    pass

LEFT = Turn('LEFT')
RIGHT = Turn('RIGHT')

Turn.finalize(locals())
```

1.1.1 Membership

Membership (if a value is a member of an enum) can be tested using the operator *in* or using *isinstance*.

Listing 4: Testing membership:

```
assert SOUTH in Direction
assert isinstance(SOUTH, Direction)

thing = object()

assert thing not in Direction
```

1.1.2 Iteration

The members of an enum type can be iterated over with the operator *in*. The order in which the members are provided by the iterator is the order of definition.

Listing 5: Iterate over enum members:

```
members = []

for member in Direction:
    members.append(member)

assert members == [NORTH, EAST, SOUTH, WEST]
```

An iterator can also be explicitely obtained using the property `member`.

Listing 6: Iterate over enum members with the property 'members':

```
members = []

for member in Direction.members:
    members.append(member)

assert members == [NORTH, EAST, SOUTH, WEST]
```

1.1.3 Sort order

Listing 7: Default ordering:

```
assert NORTH < SOUTH
assert not NORTH > SOUTH
assert EAST  > NORTH
assert not EAST < NORTH
assert EAST != NORTH
```

1.1.4 Conversion to and from numbers

Listing 8: Conversion to and from numbers:

```
assert SOUTH.ord == 2
assert int(SOUTH) == 2
assert Direction(2) is SOUTH
assert Direction[2] is SOUTH
```

1.1.5 Conversion to and from strings

Listing 9: Conversion to and from strings:

```
assert str(SOUTH) == 'SOUTH'
assert repr(SOUTH) == 'SOUTH'
assert Direction('SOUTH') is SOUTH
assert Direction['SOUTH'] is SOUTH
```

1.1.6 Explicitely specifying *ord*

`ord()` can be overridden in the default constructor with the keyword argument `ord`. It is possible to provide an own constructor, but it then should be considered whether and how to pass on `ord` to the super class *Enum*.

Listing 10: Explicitely specifying ord:

```
class Quartett(Enum):
    pass
```

(continues on next page)

(continued from previous page)

```
ONE = Quartett('ONE')                # automatic ord = 0
TWO = Quartett('TWO', ord = 100)
THREE = Quartett('THREE')            # automatic ord = 101
FOUR = Quartett('FOUR', ord = 50)

Quartett.finalize(locals())
```

Those members where `ord` is specified get the desired `ord`, but will raise an assertion if the ord value already exists. Where no `ord` is specified, an `ord` value (one) larger than all `ord` values of the members existing so far is chosen automatically. This algorithm has the advantage of preserving the definition order in the sort order as far as possible, but avoids unwelcome surprises.

Listing 11: Overridden ord values:

```
assert ONE.ord == 0
assert TWO.ord == 100
assert THREE.ord == 101
assert FOUR.ord == 50
```

The sort order is defined by the `ord` attribute of the enum members.

Listing 12: Sort order when overriding ord:

```
assert FOUR < THREE
assert FOUR < TWO
```

Whereas the iteration order is always the order in which the members where defined.

Listing 13: Unchanged iteration order when overriding ord:

```
assert list(Quartett.members) == [
    ONE, TWO, THREE, FOUR
]
```

1.1.7 Attributes

(TBD)

1.1.8 Typing

(TBD)

Listing 14: Typed function example:

```
def print_direction(d: Direction) -> None:
    print(d)


print_direction(NORTH)
```

1.2 Example(s)

The complete code examples can be found in the file *enums-example.py* in the doc distribution archive or in `doc/_build/tangle` after building the documentation with `make tangle`.

1.3 Comparison to built-in enum

- TBD: More complicated
- TBD: Only a single construction parameter

1.4 Design decisions and alternatives

(TBD)

PYTHON MODULE INDEX

INDEX