Customize and Extend Python Enum Class

Summary: in this tutorial, you'll learn how to customize and extend the custom Python enum classes.

Customize Python enum classes

Python enumerations (https://www.pythontutorial.net/python-oop/python-enumeration/) are classes (https://www.pythontutorial.net/python-oop/python-class/) . It means that you can add methods to them, or implement the dunder methods to customize their behaviors.

The following example defines the `PaymentStatus` enumeration class:

```
from enum import Enum


class PaymentStatus(Enum):
    PENDING = 1
    COMPLETED = 2
    REFUNDED = 3
```

The `PaymentStatus` enumeration has three members: `PENDING` , `COMPLETED` , and `REFUND` .

The following displays the member of the `PaymentStatus` ' member:

```
print(PaymentStatus.PENDING)
```

It shows the following:

```
PaymentStatus.PENDING
```

To customize how the `PaymentStatus` member's is represented in the string, you can implement the `__str__` (https://www.pythontutorial.net/python-oop/python-__str__/) method. For example:

```
from enum import Enum


class PaymentStatus(Enum):
    PENDING = 1
    COMPLETED = 2
    REFUNDED = 3

    def __str__(self):
        return f'{self.name.lower()}({self.value})'


print(PaymentStatus.PENDING)
```

Now, it returns the following string:

```
pending(1)
```

Implementing __eq__ method

The following attempts to compare a member of the `PaymentStatus` enum class with an integer:

```
if PaymentStatus.PENDING == 1:
    print('The payment is pending.')
```

It shows nothing because the `PaymentStatus.PENDING` is not equal to integer 1.

To allow the comparison between `PaymentStatus` member and an integer, you can implement the `__eq__` (https://www.pythontutorial.net/python-oop/python-__eq__/) method like this:

```
from enum import Enum


class PaymentStatus(Enum):
    PENDING = 1
    COMPLETED = 2
    REFUNDED = 3

    def __str__(self):
        return f'{self.name.lower()}({self.value})'

    def __eq__(self, other):
        if isinstance(other, int):
            return self.value == other

        if isinstance(other, PaymentStatus):
            return self is other

        return False


if PaymentStatus.PENDING == 1:
    print('The payment is pending.')
```

In the `__eq__` method:

- If the value to compare is an integer, it compares the value of the member with the integer.
- If the value to compare is an instance of the `PaymentStatus` enumeration, it compares the value with the member of the `PaymentStatus` member using the `is` operator.
- Otherwise, it returns `False` .

The program works as expected and returns the following output:

```
The payment is pending.
```

Implementing __lt__ method

Suppose that you want the members of the `PaymentStatus` to follow have a sort order based on their value. And you also want to compare them with an integer.

To do that, you can implement the `__lt__` method and use the `@total_ordering` decorator from the `functools` module:

```
from enum import Enum
from functools import total_ordering


@total_ordering
class PaymentStatus(Enum):
    PENDING = 1
    COMPLETED = 2
    REFUNDED = 3

    def __str__(self):
        return f'{self.name.lower()}({self.value})'

    def __eq__(self, other):
        if isinstance(other, int):
            return self.value == other
```

```
        if isinstance(other, PaymentStatus):
            return self is other

        return False

    def __lt__(self, other):
        if isinstance(other, int):
            return self.value < other

        if isinstance(other, PaymentStatus):
            return self.value < other.value

        return False


# compare with an integer
status = 1
if status < PaymentStatus.COMPLETED:
    print('The payment has not completed')

# compare with another member
status = PaymentStatus.PENDING
if status >= PaymentStatus.COMPLETED:
    print('The payment is not pending')
```

Implementing the __bool__ method

By default, all members of an enumeration are truthy. For example:

```
for member in PaymentStatus:
    print(member, bool(member))
```

To customize this behavior, you need to implement the `__bool__` method. Suppose you want the `COMPLETED` and `REFUNDED` members to be True while the `PENDING` to be `False`.

The following shows how to implement this logic:

```
from enum import Enum
from functools import total_ordering


@total_ordering
class PaymentStatus(Enum):
    PENDING = 1
    COMPLETED = 2
    REFUNDED = 3

    def __str__(self):
        return f'{self.name.lower()}({self.value})'

    def __eq__(self, other):
        if isinstance(other, int):
            return self.value == other

        if isinstance(other, PaymentStatus):
            return self is other

        return False

    def __lt__(self, other):
        if isinstance(other, int):
            return self.value < other

        if isinstance(other, PaymentStatus):
            return self.value < other.value
```

```
        return False

    def __bool__(self):
        if self is self.COMPLETED:
            return True

        return False


for member in PaymentStatus:
    print(member, bool(member))
```

The program output the following:

```
pending(1) False
completed(2) True
refunded(3) False
```

Extend Python enum classes

Python doesn't allow you to extend an enum class unless it has no member. However, this is not a limitation. Because you can define a base class that has methods but no member and then extend this base class. For example:

First, define the `OrderedEnum` base class that orders the members by their values:

```
from enum import Enum
from functools import total_ordering


@total_ordering
class OrderedEnum(Enum):
    def __lt__(self, other):
```

```
        if isinstance(other, OrderedEnum):
            return self.value < other.value
        return NotImplemented
```

Second, define the `ApprovalStatus` that extends the `OrderedEnum` class:

```
class ApprovalStatus(OrderedEnum):
    PENDING = 1
    IN_PROGRESS = 2
    APPROVED = 3
```

Third, compare the members of the `ApprovalStatus` enum class:

```
status = ApprovalStatus(2)
if status < ApprovalStatus.APPROVED:
    print('The request has not been approved.')
```

Output:

```
The request has not been approved.
```

Put it all together:

```
from enum import Enum
from functools import total_ordering


@total_ordering
class OrderedEnum(Enum):
    def __lt__(self, other):
        if isinstance(other, OrderedEnum):
            return self.value < other.value
        return NotImplemented
```

```
class ApprovalStatus(OrderedEnum):
    PENDING = 1
    IN_PROGRESS = 2
    APPROVED = 3


status = ApprovalStatus(2)
if status < ApprovalStatus.APPROVED:
    print('The request has not been approved.')
```

Summary

- Implement dunder methods to customize the behavior of Python enum classes.
- Define an emum class with no members and methods and extends this base class.