Python Enum aliases & @enum.unique Decorator

Summary: in this tutorial, you'll learn about enumeration member aliases and how to use the enum unique decorator to ensure the uniqueness of member values.

Introduction to the enum aliases

By definition, the enumeration (https://www.pythontutorial.net/python-oop/python-enumeration/) member values are unique. However, you can create different member names with the same values.

For example, the following defines the `Color` enumeration:

```
from enum import Enum


class Color(Enum):
    RED = 1
    CRIMSON = 1
    SALMON = 1
    GREEN = 2
    BLUE = 3
```

In this example, the `Color` enumeration has the `RED`, `CRIMSON`, and `SALMON` members with the same value 1.

When you define multiple members in an enumeration with the same values, Python does not create different members but aliases.

In this example, the `RED` is the main member while the `CRIMSON` and `SALMON` members are the aliases of the `RED` member

The following statements return `True` because `CRIMSON` and `SALMON` members are RED meber:

```
print(Color.RED is Color.CRIMSON)
print(Color.RED is Color.SALMON)
```

Output:

```
True
True
```

When you look up a member by value, you'll always get the main member, not aliases. For example, the following statement returns the `RED` member:

```
print(Color(1))
```

Output:

```
Color.RED
```

When you iterate the members of an enumeration with aliases, you'll get only the main members, not the aliases. For example:

```
for color in Color:
    print(color)
```

It returns only three members:

```
Color.RED
Color.GREEN
Color.BLUE
```

To get all the members including aliases, you need to use the `__member__` property of the enumeration class. For example:

```
from enum import Enum
from pprint import pprint


class Color(Enum):
    RED = 1
    CRIMSON = 1
    SALMON = 1
    GREEN = 2
    BLUE = 3


pprint(Color.__members__)
```

Output:

```
mappingproxy({'BLUE': <Color.BLUE: 3>,
              'CRIMSON': <Color.RED: 1>,
              'GREEN': <Color.GREEN: 2>,
              'RED': <Color.RED: 1>,
              'SALMON': <Color.RED: 1>})
```

As shown clearly from the output, the `CRIMSON` and `SALMON` reference the same object which is referenced by the `RED` member:

```
<Color.RED: 1>
```

When to use enum aliases

Enumeration aliases can be helpful in some situations. For example, suppose that you have to deal with API from two different systems. And each system has a different response status doe with the same meaning as shown in the following table:

System 1	System 2	Meaning
REQUESTING	PENDING	The request is in progress
OK	FULFILLED	The request was completed successfully
NOT_OK	REJECTED	The request was failed

To standardize the status codes from these systems, you can use enumeration aliases as follows:

Your System	System 1	System 2	Meaning
IN_PROGRESS	REQUESTING	PENDING	The request is in progress
SUCCESS	OK	FULFILLED	The request was completed successfully
ERROR	NOT_OK	REJECTED	The request was failed

The following defines the `ResponseStatus` enumeration with aliases:

```
from enum import Enum


class ResponseStatus(Enum):
    # in progress
    IN_PROGRESS = 1
```

```
    REQUESTING = 1
    PENDING = 1

    # success
    SUCCESS = 2
    OK = 2
    FULFILLED = 2

    # error
    ERROR = 3
    NOT_OK = 3
    REJECTED = 3
```

The following compares the response code from system 1 to check if the request was successful or not:

```
code = 'OK'
if ResponseStatus[code] is ResponseStatus.SUCCESS:
    print('The request completed successfully')
```

Output:

```
The request completed successfully
```

Likewise, you can check the response code from system 2 to see if the request was successful:

```
code = 'FULFILLED'
if ResponseStatus[code] is ResponseStatus.SUCCESS:
    print('The request completed successfully')
```

Output:

```
print('The request completed successfully')
```

@enum.unique decorator

To define an enumeration with no aliases, you can carefully use unique values for the members. For example:

```
from enum import Enum


class Day(Enum):
    MON = 'Monday'
    TUE = 'Tuesday'
    WED = 'Wednesday'
    THU = 'Thursday'
    FRI = 'Friday'
    SAT = 'Saturday'
    SUN = 'Sunday'
```

But you can accidentally use the same values for two members like this:

```
class Day(Enum):
    MON = 'Monday'
    TUE = 'Monday'
    WED = 'Wednesday'
    THU = 'Thursday'
    FRI = 'Friday'
    SAT = 'Saturday'
    SUN = 'Sunday'
```

In this example, the `TUE` member is the alias of the `MON` member, which you may not expect.

To ensure an enumeration has no alias, you can use the `@enum.unique` decorator from the `enum` module.

When you decorate an enumeration with the `@enum.unique` decorator, Python will throw an exception if the enumeration has aliases.

For example, the following will raise a `ValueError`:

```
import enum

from enum import Enum


@enum.unique
class Day(Enum):
    MON = 'Monday'
    TUE = 'Monday'
    WED = 'Wednesday'
    THU = 'Thursday'
    FRI = 'Friday'
    SAT = 'Saturday'
    SUN = 'Sunday'
```

Error:

```
ValueError: duplicate values found in <enum 'Day'>: TUE -> MON
```

Summary

- When an enumeration has different members with the same values, the first member is the main member while others are aliases of the main member.
- Use the `@enum.unique` decorator from the `enum` module to enforce the uniqueness of the values of the members.