Take Advantage of the Enum Class to Implement Enumerations in Python

Create your own enumerations easily

Yong Cui Follow
Jul 30, 2020 · 7 min read ★

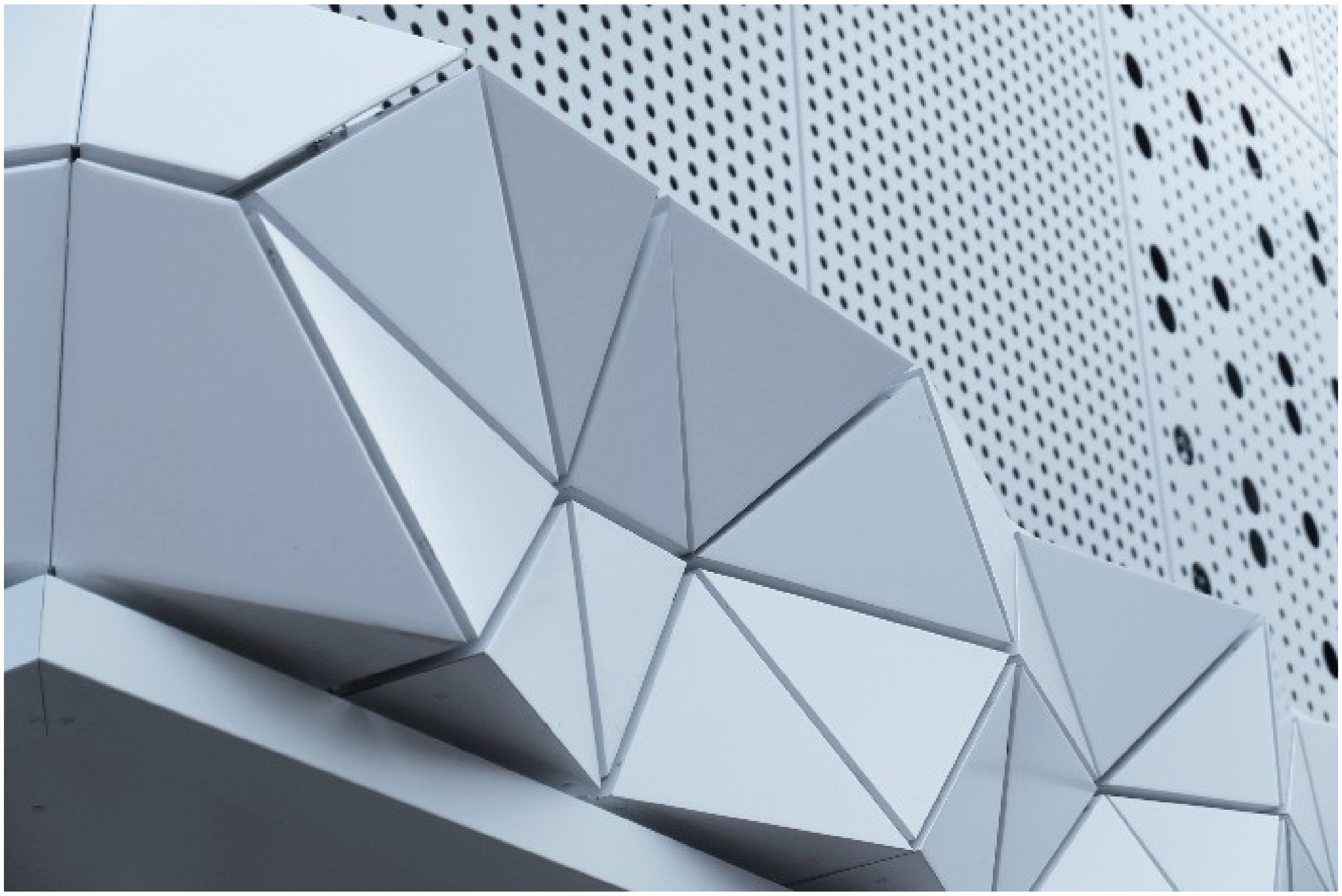

Photo by Oleg Laptev on Unsplash.

enumerations are how you can organize a group of closely related members under the same umbrella.

The enumeration written in Kotlin below shows you that we organized four directions under the concept of `Direction` , with each member being a direction:

```
enum class Direction {
    NORTH,
    EAST,
    SOUTH,
    WEST
}
```

Enumeration_Direction.kt hosted with ♡ by GitHub view raw

Enumeration of Direction

In addition to directions, colors, nationalities, seasons, weekdays, and many other kinds of information that have discrete members can all be implemented as enumerations. As you may know, enumerations give us the benefit of easier handling of related concepts. Thus, we should consider enumerations when we encounter these data.

In most of these programming languages, enumerations are usually conveyed as a basic form of data structures, and these topics appear early in one's learning journey. However, for some reason, it's less discussed when people are learning Python. I don't know the exact reasons, but one possible reason is that in Python, enumeration is not a built-in data structure (unlike `bool` , `str` , `int` , and `list`). Thus, some people have attempted to mimic the behaviors of enumeration using a custom class, as shown below:

```
# Define a class
class DataStatus:
    SUCCEEDED = 1
    ERROR = 2

# Use the members
DataStatus.SUCCEEDED
DataStatus.ERROR
```

However, the implementation of enumeration with a regular class doesn't have all the features that you may think of with enumerations in other programming languages. One such problem is shown below. In essence, such an implementation is "hacking" by utilizing the attributes of a Python class. Thus, if you check the type, you'll find that it doesn't behave ideally. In this case, the type is to be `int`, which reveals the type of the attribute. In other words, they're not treated as members of a certain group. Instead, they're simply separate attributes of the class:

```
>>> # Introspection
... print("Type Checking:", type(DataStatus.SUCCEEDED))
... print("Check Instance Type:", isinstance(DataStatus.SUCCEEDED, DataStatus))
...
Type Checking: <class 'int'>
Check Instance Type: False
```

enumeration_class_problems.py hosted with ♡ by GitHub view raw

Potential problems of enumerations using a regular class

In this article, we're going to learn the proper way to define a real working enumeration class by taking advantage of the `enum` module, which is part of Python's standard library. So once you've installed Python on your computer, it's ready to use the `enum` module for enumeration creation.

Declare an Enumeration Class

Declaring an enumeration class is actually super easy in Python. To do that, you just create your own class by subclassing the `Enum` class from the `enum` module, as shown in the code snippet below:

```
from enum import Enum

class Direction(Enum):
```

```python
    WEST = 4

class DirectionOneLiner(Enum):
    NORTH = 1; EAST = 2; SOUTH = 3; WEST = 4
```

enumeration_direction.py hosted with ♡ by GitHub view raw

Enumeration of Direction in Python

The syntax is pretty straightforward. In the class definition, we list the members and their associated values — much like listing them as attributes. The code above also shows you that you can declare your members by using semicolons to separate the members in a single line, although I recommend the former style of defining one member per line because it will enhance your code's readability.

Although we usually care less about the raw values of these enumerated members, one thing to note is that you don't have to use integers for their raw values. For instance, you can use strings if you prefer. In addition, when we use integers, these numbers don't have to be incremental. Here's some simple code to show you these features:

```python
class DirectionRandomValue(Enum):
    NORTH = 111
    EAST = 222
    SOUTH = 333
    WEST = 444


class DirectionString(Enum):
    NORTH = 'N'
    EAST = 'E'
    SOUTH = 'S'
    WEST = 'W'
```

enumeration_raw_values.py hosted with ♡ by GitHub view raw

Flexibility of raw values

enumeration class. However, the `Enum` class is different from regular classes. These attributes are considered as members of the enumeration, as they're supposed to be. With this distinction, we can use the introspection functions (e.g. `type` and `isinstance`). Some related code is shown below:

```
>>> print(Direction.NORTH)
... print("Type:", type(Direction.SOUTH))
... print("Check Instance:", isinstance(Direction.EAST, Direction))
...
Direction.NORTH
Type: <enum 'Direction'>
Check Instance: True
```

enumeration_members.py hosted with ♡ by GitHub view raw

Use Members

As you can see, these "attributes" of the enumeration class are of the type of the class. When we check the instance, the object's class is indeed the enumeration class — `Direction`. They work exactly as we expected with an enumeration class.

For each member, it has additional attributes. The useful ones include `name` and `value`, which represent the enumerated member's name and its associated value, integer, or string if you define it as such.

Name and value

By understanding the value of the member, we can do something interesting with it. For instance, suppose that we get a JSON object response from a web HTTP request that uses a numeric value (e.g. 2) to denote the direction. With this integer, we can create a `Direction` instance member. Once it's created, it's easy to use the member for various other operations:

Member creation with raw value

As shown above, we created a `Direction` member using the fetched value of 2. With the created instance, we can make extra-simple comparisons, and as expected, the fetched direction is evaluated to be `True` when it's compared with the east direction. Notably, we use the identity comparison here by using the `is` keyword because you can think of members as singleton instance objects of the enumeration class. However, you'll still get valid comparison results if you use equality comparisons using `==`, which compares the values.

As a side note, if you're trying to create a member from a value that isn't valid, (e.g. > 4 in our example), you'll encounter an exception, as shown below:

Value error

Iterate Members

We often need to iterate all the members of the enumeration class. The following code shows you the most basic form of iteration with the enumeration class:

For loop

As you can see, we could simply use the class name in a for loop, which indicates that the enumeration class itself is an iterable (i.e. a Python object that can be iterated). Thus, we can also create a list of members from the class name because it is an iterable.

List of directions

Other advanced knowledge related to this iteration is using the special method `__members__`, which will retrieve the mapping of the enumeration class with the names and their members stored as key-value pairs. Please note that mapping is not exactly the same concept as a dictionary, although they sound similar. Mapping is a more general term than dictionaries.

Items of members

Define Methods

Although the base `Enum` class is different from other regular classes because it specially handles its attributes as members, we can still define methods for the class like other classes. This gives us the opportunity to have more powerful and flexible operations with the enumeration class.

Suppose that we want to find out the angles of the directions in a map. We can implement some related functionalities, as shown below:

Custom methods

As shown above, the `angle()` function is defined as an instance method that will calculate the degrees of a certain direction, with the north being 0 degrees. To show you a proof of concept, I also declared a static method `angle_interval()` that calculates the difference in angles between two directions. The usage of these methods is shown in examples following the declared class in the code snippet above.

Functional APIs

The previous sections reviewed the class-based implementation of the enumeration. To make `Enum` more flexible, Python provides functional APIs for the purpose of creating enumeration. If you don't know what a functional API is, here's an intuitive explanation. In our previous sections, we created enumeration members through a definition of a subclass of the `Enum` class. As you may know, it's mostly an object-oriented programming (OOP) style. Basically, we use classes and objects to handle members. By contrast, the functional APIs take a functional approach. We create enumerations simply by calling a function.

Still confusing? Let's see some related code first. To make the comparisons more straightforward, I'll implement the same direction members using the functional API approach. As shown below, we'll just use the `Enum` function — which should be termed as a factory function, to be more precise. Like the `namedtuple` function, these kinds of functions create a new class:

Enumeration with functional API

Because we didn’t use a class declaration interface to implement this enumeration class, is it possible to define functions for this enumeration class (the one created using the functional API)? The answer is yes because we can take advantage of a feature called monkey patching in Python. Simply put, it’s a technique that allows you to create and/or modify the behavior of existing modules or classes during runtime. Let’s see a quick example below:

Monkey patching

As shown above, we first created a function and assigned the function to the `DirectionFunctional` class's `angle` attribute. Notably, this attribute is considered an instance method because the `angle` function's first argument is the class name. It's the same as the self argument in the function declared in the class.

After the monkey patching, we were able to get the angle for the south direction, which is 180 degrees, as expected.

are mainly achieved by taking advantage of creating a subclass of the `Enum` class that is in the `enum` module. Here's a quick recap of the key points.

- Unlike regular classes, attributes of the `Enum` class will be considered independent members of the class.
- These members have names and values, which can be integers (they're the default ones), strings, or other data types.
- Methods can be declared with the enumeration class, which allows us to create additional operations for easier handling of the members.
- We can iterate all the members of the class, which is an iterable itself. We can also create a list of these members by taking advantage of the class being an iterable.
- As a flexible feature, we can use functional APIs to create an enumeration class during runtime. In addition, we can use monkey patching to add additional methods to the class.

Thanks for reading this piece. If you want to read more about enumerations in Python, you can refer to the official documentation.

Thanks to Zack Shapiro.

Python Programming Artificial Intelligence Data Science Technology