The Addition of the Enum type to the Python Standard Library

Group Members: Jonathan Song, Gabriella Mendoza, Thomas Wagner

Introduction:

This proposed change to Python concerns the addition of the Enum type to the standard library. The addition of an enumeration type allows programmers to create sets of symbolic names that are bound to unique, constant values. This Enum type will ensure that each enumeration member is distinct from each other in name and in value and provides the programmer with unique, built-in methods to iterate over members and present each enumeration in a printable representation.

How does it apply to terminology and concepts used throughout the course?

This proposed change reflects concepts such as type matching, immutable variables, and alias members, which we have covered in this course.

When adding enumeration members to an Enum type, each of the members must match the type of the enumeration itself. If not, then use of the enumeration may result in an error, as it will not pass the qualifications of the basic type checking form that object of Enum type. In discussions over Python mail, it was decided by Guido van Rossum and other members of the community that all enumeration members must belong to the type of their Enum in order to make them behave in “a more intuitive way” by having it take on more properties similar to the bool type of Python. For example, it is ensured that the types of members such as Direction.up will always be of type Direction. [1]

Python is a dynamically typed language, which means that the type checking of the program occurs as the program is being evaluated. Therefore, it is possible for an enumeration member to be incorrectly defined without being caught as long as it is not used. These specifications of the Enum type offer type safety to its use in the language by ensuring that the programs that are run are well-typed. When we say that an expression is well-typed, we are saying that in the particular environment, the expression follows the rules that are specified in that environment and can therefore have a type inferred from it. In cases where the enumeration members do not follow the rules specified in their particular type environments, which are specified by their Enum types, we will say that the expressions are not well typed and we throw a typeerror to stop the program from evaluating those values any further. In the Scala interpreter of JavaScript built in class, this is similar to the use of the “typeerror” token that was used to represent the occurrence of a dynamic type error if it should encounter some ill-typed or not well-typed expression. It will indicate that there has been a violation of some predefined typing rules and will not allow the program to finish evaluating due to this error.

In addition, the Enum type is used to define unique, related sets of constant values. Another way of stating this is that when enumeration members are defined, they become immutable variables. This means that when a variable is defined and is set to value, the variable cannot be re-set to a different value in the same environment. As a result, there cannot be two Enum members that have the same name, as this would potentially serve as a violation of the immutable property of the Enum type.

For example, something like this:

```
from enum import Enum

class Color(Enum):
    red = 0
    white = 1
    black = 2
    black = 3
```

Will result in a type error, because there are two enumeration members with the same name "black". The enumeration member "black" has already been declared and assigned to the value of 2, and will not be allowed to be re-assigned to the value of 3 in the manner shown above. This property of the enumeration type in Python demonstrates the safety of the enumeration members through their immutability.

Also, the addition of the Enum type to the Python standard library supports the concept of creating alias members. If two Enum members should be assigned to the same value, then an alias is created:

```
from enum import Enum

class Color(Enum):
   red = 1
   white = 2
   black = 2
```

In this case, members "white" and "black" are both assigned to the same value of "2". Within the enumeration type, such an operation is not valid and "white" and "black" cannot exist as independent members of the same enumeration in this case. Instead, the second variable that was created that points to the same value as the first variable that points to the same value will become an alias. Therefore, "black" will become an alias to "white." In practice, this means that lookup of variable "black" will return "white," which is set to the value of 2. The concept of alias members is reminiscent of the concepts of pointers and values that are defined in other languages such as in C or C++. In this case, the member "black" becomes like a pointer to the member "white," and will return that member when it is called rather than returning the value of 2 immediately.

Because of this, the above code can also be rewritten as:

```
from enum import Enum

class Color(Enum):
   red = 1
   white = 2
   alias_for_white = 2
```

The alias members of an enumeration are treated differently than the normal members of an enum in that iteration over the enum members will not provide them without the use of a special attribute “**members**”. In addition, lookup of the alias members will return the enumeration member that it is pointing to, as was described above.

Is this change useful?

The addition of the Enum type was first proposed in PEP 354, but was ultimately rejected due to lack of interest. The claim at this time was that the addition of the Enum type was not needed, as the Python standard library of the time had plenty of “individual data structures,” “cookbook recipes, and PyPI packages” that have already met the needs of the enumeration type. [2]

For example, members of the community such as Guido van Rossum proposed the addition of the simple flufl.enum package, which was a very basic implementation of the Enum library that is currently implemented in the standard library due to the belief that “enums are not that useful in small programs.” It was believed that the extra cost of defining the enum type was more than it was worth, leading them to use this very basic enum type in the standard library for a time. [3]

Before the introduction of the Enum type, enumerations could be pseudo-implemented using classes:

```
class Directions:
    up = 0
    down = 1
    left = 2
    right =3
```

[4]

Here, we have created a class with particular fields that acts like an enumeration. In fact, we can use this class to do things that we might want to do with the enum type, which may suggest that it is not needed:

```
if direction == Direction.up:
   return true
else:
   return false
```

[5]

In that case, why do we bother talking about the addition of the Enum type to the standard library? The reason for this lies in the convenience of the programmer. The Enum type ensures that the values of each enumeration member are each distinct from each other, and also makes sure that this property holds for members of other enumerations. In addition, the Enum type provides a convenient prebuilt method for the programmer to iterate over the enumeration members and also gives them the “ability to have named constants” that they can use.

In events such as PyCon, several people expressed interest in the addition of a more formal enumeration type in the standard library that would allow them to create a subclass of int. This would allow them to replace integer constants in the standard library with more friendly string representations that would be easy to use “without ceding backward compatibility.” This particular feature came to be in the form of the IntEnum subclass of the Enum type, which allowed for the comparison of integers and ultimately provide “greater interoperability with integers.” [6]

Changing the above Direction class to an Enum type is simple and does not require much more code:

```
from enum import Enum

class Directions(Enum):
    up = 0
    down = 1
    left = 2
    right = 3
```

As can be seen, the only changes that needed to be made to the previous code is the inclusion of the enum module and passing of the Enum type as a parameter to the Directions class.

Basic use

The Enum type can be brought into a program for us to use by importing in the enum module:

```
from enum import Enum
```

Once this is done, a programmer is able to call the Enum class, which is generally defined as: Enum(name of enumeration, source of enumeration member names). The first parameter of this call is simply the name of the enumeration, and can be used for calls in a program. The second parameter of the call encompasses the members of the enumeration, and can be defined in a variety of ways. For example, the members can be imported as a name sequence, a tuple sequence, a mapping, or a string.

A call such as this is of the form:

```
Animals = Enum('Animals', 'ant bee cat dog', module = __name__)
```

In this case, the enumeration is called "Animals" and consists of members "ant," "bee," "cat," and "dog," which were passed in as a whitespace-separated string.

Here is an example of the creation an enum type. This should look familiar:

```
from enum import Enum

class Directions(Enum):
    up = 0
    down = 1
    left = 2
    right = 3
```

The IntEnum subclass is defined similarly to the Enum class and allows comparisons with integer types:

```
from enum import IntEnum

class VerticalDirections(IntEnum):
    up = 1
   down = 2

class HorizontalDirections(IntEnum):
   left = 1
   right = 2

VerticalDirections(up) == 1
```

Once the integer enumerations are defined, the programmer may use equality operations between enumeration members and integer type values, which greatly increases the extent to which the systems can exchange information.

In addition, the Enum type is undergoing more proposed changes to provide convenience to the programmer by allowing them to pass in enumeration members without assigning them to values right away:

```
from enum import Enum

class Directions(Enum):
    up, down, left, right
```

In this case, the enumeration members do not have values assigned to them, so values are automatically assigned to them upon lookup. This saves some of the programmer's time as it saves them from some typing when performing repetitive tasks.

Conclusion

The formal addition of this Enum type to the standard library has allowed Python to evolve into a more convenient language for programmers to use by providing them with prebuilt methods to create a set of well-typed, immutable varaiables that the programmer may iterate over and perform other actions with. The Enun type has many features that ensures the safety of the user in its usage and creation of enumeration members, which applies concepts such as type checking and alias members to validate the correct usage of the type. In spite of the original resistance of people to the additon of the enum module, the addition of the Enum type has proved very useful in the way that it has increased the interoperability of with many variable types and the convenience of the programmer.

References

[1] https://mail.python.org/pipermail/python-dev/2013-April/125687.html

[2] https://www.python.org/dev/peps/pep-0354/

[3] https://mail.python.org/pipermail/python-ideas/2013-February/019373.html

[4] http://stackoverflow.com/questions/1969005/enumerations-in-python

[5] http://stackoverflow.com/questions/7886958/no-need-for-enums

[6] https://www.python.org/dev/peps/pep-0435/