Build Enumerations of Constants With Python's Enum

by Leodanis Pozo Ramos Oct 03, 2022 3 Comments

intermediate python

Table of Contents

Some programming languages, like Java and C++, include syntax that supports a data type known as **enumerations**, or just **enums**. This data type allows you to create sets of semantically related constants that you can access through the enumeration itself. Python doesn't have a dedicated syntax for enums. However, the Python standard library has an `enum` module that supports enumerations through the `Enum` class.

If you come from a language with enumerations, and you're used to working with them, or if you just want to learn how to use enumerations in Python, then this tutorial is for you.

In this tutorial, you'll learn how to:

- Create **enumerations** of constants using Python's `Enum` class
- Work with enumerations and their **members** in Python
- Customize enumeration classes with **new functionalities**
- Code **practical examples** to understand why you would use enumerations

Additionally, you'll explore other specific enumeration types that live in `enum`, including `IntEnum`, `IntFlag`, and `Flag`. They'll help you create specialized enums.

To follow along with this tutorial, you should be familiar with object-oriented programming and inheritance in Python.

> **Source Code: Click here to download the free source code** that you'll use to build enumerations in Python.

Getting to Know Enumerations in Python

Several programming languages, including Java and C++, have a native **enumeration** or **enum** data type as part of their syntax. This data type allows you to create sets of named constants, which are considered **members** of the containing enum. You can access the members through the enumeration itself.

Enumerations come in handy when you need to define an immutable and discrete set of similar or related constant values that may or may not have semantic meaning in your code.

Days of the week, months and seasons of the year, Earth's cardinal directions, a program's status codes, HTTP status codes, colors in a traffic light, and pricing plans of a web service are all great examples of enumerations in programming. In general, you can use an enum whenever you have a variable that can take one of a *limited set of possible values*.

Python doesn't have an enum data type as part of its syntax. Fortunately, Python 3.4 added the `enum` module to the standard library. This module provides the `Enum` class for supporting general-purpose enumerations in Python.

Enumerations were introduced by PEP 435, which defines them as follows:

>

> An enumeration is a set of symbolic names bound to unique, constant values. Within an enumeration, the values can be compared by identity, and the enumeration itself can be iterated over. (Source)

Before this addition to the standard library, you could create something similar to an enumeration by defining a sequence of similar or related constants. To this end, Python developers often used the following idiom:

Python >>>

```
>>> RED, GREEN, YELLOW = range(3)

>>> RED
0

>>> GREEN
1
```

Even though this idiom works, it doesn't scale well when you're trying to group a large number of related constants. Another inconvenience is that the first constant will have a value of `0`, which is falsy in Python. This can be an issue in certain situations, especially those involving Boolean tests.

> **Note:** If you're using a Python version before 3.4, then you can create enumerations by installing the enum34 library, which is a backport of the standard-library `enum`. The aenum third-party library could be an option for you as well.

In most cases, enumerations can help you avoid the drawbacks of the above idiom. They'll also help you produce more organized, readable, and robust code. Enumerations have several benefits, some of which relate to ease of coding:

- Allowing for conveniently **grouping related constants** in a sort of namespace
- Allowing for **additional behavior** with custom methods that

operate on either enum members or the enum itself

- Providing quick and flexible **access** to enum members
- Enabling **direct iteration** over members, including their names and values
- Facilitating **code completion** within IDEs and editors
- Enabling **type** and **error checking** with static checkers
- Providing a hub of **searchable** names
- Mitigating **spelling mistakes** when using the members of an enumeration

They also make your code robust by providing the following benefits:

- Ensuring **constant values** that can't be changed during the code's execution
- Guaranteeing **type safety** by differentiating the same value shared across several enums
- Improving **readability** and **maintainability** by using descriptive names instead of mysterious values or magic numbers
- Facilitating **debugging** by taking advantage of readable names instead of values with no explicit meaning
- Providing a **single source of truth** and **consistency** throughout the code

Now that you know the basics of enumerations in programming and in Python, you can start creating your own enum types by using Python's `Enum` class.

Creating Enumerations With Python's `Enum`

Python's `enum` module provides the `Enum` class, which allows you to create enumeration types. To create your own enumerations, you can either subclass `Enum` or use its functional API. Both options will let you define a set of related constants as enum members.

In the following sections, you'll learn how to create enumerations in your code using the `Enum` class. You'll also learn how to set automatically generated values for your enums and how to create enumerations containing alias and unique values. To kick things off, you'll start by learning how to create an enumeration by subclassing `Enum`.

Creating Enumerations by Subclassing `Enum`

The `enum` module defines a general-purpose enumeration type with iteration and comparison capabilities. You can use this type to create sets of named constants that you can use to replace literals of common data types, such as numbers and strings.

A classic example of when you should use an enumeration is when you need to create a set of enumerated constants representing the days of the week. Each day will have a symbolic name and a numeric value between `1` and `7`, inclusive.

Here's how you can create this enumeration by using `Enum` as your **superclass** or **parent class**:

Python >>>

```
>>> from enum import Enum

>>> class Day(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...
```

```
>>> list(Day)
[
    <Day.MONDAY: 1>,
    <Day.TUESDAY: 2>,
    <Day.WEDNESDAY: 3>,
    <Day.THURSDAY: 4>,
    <Day.FRIDAY: 5>,
    <Day.SATURDAY: 6>,
    <Day.SUNDAY: 7>
]
```

Your `Day` class is a subclass of `Enum`. So, you can call `Day` an **enumeration**, or just an **enum**. `Day.MONDAY`, `Day.TUESDAY`, and the like are **enumeration members**, also known as **enum members**, or just **members**. Each member must have a **value**, which needs to be constant.

Because enumeration members must be constants, Python doesn't allow you to assign new values to enum members at runtime:

Python >>>

```
>>> Day.MONDAY = 0
Traceback (most recent call last):
    ...
AttributeError: Cannot reassign members.

>>> Day
<enum 'Day'>

>>> # Rebind Day
>>> Day = "Monday"
>>> Day
'Monday'
```

If you try to change the value of an enum member, then you get an `AttributeError`. Unlike member names, the name containing the enumeration itself isn't a constant but a variable. So, it's possible to rebind this name at any moment during your program's execution, but you should avoid doing that.

In the example above, you've reassigned `Day`, which now holds a string rather than the original enumeration. By doing this, you've

lost the reference to the enum itself.

Often, the values mapped to members are consecutive integer numbers. However, they can be of any type, including user-defined types. In this example, the value of `Day.MONDAY` is `1`, the value of `Day.TUESDAY` is `2`, and so on.

> **Note**: You may have noticed that the members of `Day` are capitalized. Here's why:
>
> > Because Enums are used to represent constants we recommend using UPPER_CASE names for enum members… (Source)

You can think of enumerations as collections of constants. Like lists, tuples, or dictionaries, Python enumerations are also iterable. That's why you can use `list()` to turn an enumeration into a `list` of enumeration members.

The members of a Python enumeration are instances of the container enumeration itself:

Python >>>

```
>>> from enum import Enum

>>> class Day(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...

>>> type(Day.MONDAY)
<enum 'Day'>

>>> type(Day.TUESDAY)
<enum 'Day'>
```

You shouldn't confuse a custom enum class like `Day` with its members: `Day.MONDAY`, `Day.TUESDAY`, and so on. In this example, the `Day` enum type is a hub for enumeration members, which happen to be of type `Day`.

You can also use the idiom based on `range()` to build enumerations:

Python >>>

```
>>> from enum import Enum

>>> class Season(Enum):
...     WINTER, SPRING, SUMMER, FALL = range(1, 5)
...

>>> list(Season)
[
    <Season.WINTER: 1>,
    <Season.SPRING: 2>,
    <Season.SUMMER: 3>,
    <Season.FALL: 4>
]
```

In this example, you use `range()` with the `start` and `stop` offsets. The `start` offset allows you to provide the number that starts the range, while the `stop` offset defines the number at which the range will stop generating numbers.

Even though you use the `class` syntax to create enumerations, they're special classes that differ from normal Python classes. Unlike regular classes, enums:

- Can't be instantiated
- Can't be subclassed unless the base enum has no members
- Provide a human-readable string representation for their members
- Are iterable, returning their members in a sequence
- Provide hashable members that can be used as dictionary keys
- Support the **square bracket** syntax, call syntax, and **dot notation** to access members
- Don't allow member reassignments

You should keep in mind all these subtle differences when you start creating and working with your own enumerations in Python.

Often, the members of an enumeration take consecutive integer values. However, in Python, the values of members can be of any type, including user-defined types. For example, here's an enumeration of school grades that uses non-consecutive numeric values in descending order:

Python >>>

```
>>> from enum import Enum

>>> class Grade(Enum):
...     A = 90
...     B = 80
...     C = 70
...     D = 60
...     F = 0
...

>>> list(Grade)
[
    <Grade.A: 90>,
    <Grade.B: 80>,
    <Grade.C: 70>,
    <Grade.D: 60>,
    <Grade.F: 0>
]
```

This example shows that Python enums are pretty flexible and allow you to use any meaningful value for their members. You can set the member values according to the intent of your code.

You can also use string values for your enumeration members. Here's an example of a `Size` enumeration that you can use in an online store:

Python >>>

```
>>> from enum import Enum

>>> class Size(Enum):
...     S = "small"
```

```
...     M = "medium"
...     L = "large"
...     XL = "extra large"
...

>>> list(Size)
[
    <Size.S: 'small'>,
    <Size.M: 'medium'>,
    <Size.L: 'large'>,
    <Size.XL: 'extra large'>
]
```

In this example, the value associated with each size holds a description that can help you and other developers understand the meaning of your code.

You can also create enumerations of Boolean values. In this case, the members of your enumeration will have only two values:

Python >>>

```
>>> from enum import Enum

>>> class SwitchPosition(Enum):
...     ON = True
...     OFF = False
...

>>> list(SwitchPosition)
[<SwitchPosition.ON: True>, <SwitchPosition.OFF: False>]

>>> class UserResponse(Enum):
...     YES = True
...     NO = False
...

>>> list(UserResponse)
[<UserResponse.YES: True>, <UserResponse.NO: False>]
```

These two examples show how you can use enumerations to add extra context to your code. In the first example, anyone reading your code will know that the code emulates a switch object with two possible states. This additional information highly improves your

code's readability.

You can also define an enumeration with heterogeneous values:

Python >>>

```
>>> from enum import Enum

>>> class UserResponse(Enum):
...     YES = 1
...     NO = "No"
...

>>> UserResponse.NO
<UserResponse.NO: 'No'>

>>> UserResponse.YES
<UserResponse.YES: 1>
```

However, this practice makes your code inconsistent from a type safety perspective. Therefore, it's not recommended practice. Ideally, it would help if you had values of the same data type, which is consistent with the idea of grouping similar, related constants in enumerations.

Finally, you can also create empty enumerations:

Python >>>

```
>>> from enum import Enum

>>> class Empty(Enum):
...     pass
...

>>> list(Empty)
[]

>>> class Empty(Enum):
...     ...
...

>>> list(Empty)
[]

>>> class Empty(Enum):
```

```
...     """Empty enumeration for such and such purposes."""
...

>>> list(Empty)
[]
```

In this example, `Empty` represents an empty enumeration because it doesn't define any member constants. Note that you can use the `pass` statement, the `Ellipsis` literal (`...`), or a class-level docstring to create empty enumerations. This last approach can help you improve the readability of your code by providing extra context in the docstring.

Now, why would you need to define an empty enumeration anyway? Empty enumerations can come in handy when you need to build a hierarchy of enum classes to reuse functionality through inheritance.

Consider the following example:

Python >>>

```
>>> from enum import Enum
>>> import string

>>> class BaseTextEnum(Enum):
...     def as_list(self):
...         try:
...             return list(self.value)
...         except TypeError:
...             return [str(self.value)]
...

>>> class Alphabet(BaseTextEnum):
...     LOWERCASE = string.ascii_lowercase
...     UPPERCASE = string.ascii_uppercase
...

>>> Alphabet.LOWERCASE.as_list()
['a', 'b', 'c', 'd', ..., 'x', 'y', 'z']
```

In this example, you create `BaseTextEnum` as an enumeration with no members. You can only subclass a custom enumeration if it doesn't

have members, so `BaseTextEnum` qualifies. The `Alphabet` class inherits from your empty enumeration, which means that you can access the `.as_list()` method. This method converts the value of a given member into a list.

Creating Enumerations With the Functional API

The `Enum` class provides a functional API that you can use to create enumerations without using the usual class syntax. You'll just need to call `Enum` with appropriate arguments like you'd do with a function or any other callable.

This functional API resembles the way in which the `namedtuple()` factory function works. In the case of `Enum`, the functional signature has the following form:

Python

```
Enum(
    value,
    names,
    *,
    module=None,
    qualname=None,
    type=None,
    start=1
)
```

From this signature, you can conclude that `Enum` needs two positional arguments, `value` and `names`. It can also take up to four optional and keyword-only arguments. These arguments are `module`, `qualname`, `type`, and `start`.

Here's a table that summarizes the content and meaning of each argument in the signature of `Enum`:

Argument	Description	Required
`value`	Holds a string with the name of the new enumeration class	Yes
`names`	Provides names for the enumeration members	Yes
`module`	Takes the name of the module that defines the enumeration class	No
`qualname`	Holds the location of the module that defines the enumeration class	No
`type`	Holds a class to be used as the first mixin class	No
`start`	Takes the starting value from the enumeration values will begin	No

To provide the `names` argument, you can use the following objects:

- A string containing member names separated either with spaces or commas
- An iterable of member names
- An iterable of name-value pairs

The `module` and `qualname` arguments play an important role when you need to pickle and unpickle your enumerations. If `module` isn't set, then Python will attempt to find the module. If it fails, then the class will not be picklable. Similarly, if `qualname` isn't set, then Python will set it to the global scope, which may cause your enumerations to fail unpickling in some situations.

The `type` argument is required when you want to provide a mixin class for your enumeration. Using a mixin class can provide your custom enum with new functionality, such as extended comparison

capabilities, as you'll learn in the section about mixing enumerations with other data types.

Finally, the `start` argument provides a way to customize the initial value of your enumerations. This argument defaults to `1` rather than to `0`. The reason for this default value is that `0` is false in a Boolean sense, but enum members evaluate to `True`. Therefore, starting from `0` would seem surprising and confusing.

Most of the time, you'll just use the first two arguments to `Enum` when creating your enumerations. Here's an example of creating an enumeration of common HTTP methods:

Python >>>

```
>>> from enum import Enum

>>> HTTPMethod = Enum(
...     "HTTPMethod", ["GET", "POST", "PUSH", "PATCH", "DELETE"
... )

>>> list(HTTPMethod)
[
    <HTTPMethod.GET: 1>,
    <HTTPMethod.POST: 2>,
    <HTTPMethod.PUSH: 3>,
    <HTTPMethod.PATCH: 4>,
    <HTTPMethod.DELETE: 5>
]
```

This call to `Enum` returns a new enumeration called `HTTPMethod`. To provide the member names, you use a list of strings. Each string represents an HTTP method. Note that the member values are automatically set to consecutive integer numbers starting from `1`. You can change this initial value using the `start` argument.

Note that defining the above enumerations with the class syntax will produce the same result:

Python >>>

```
>>> from enum import Enum

>>> class HTTPMethod(Enum):
```

```
...     GET = 1
...     POST = 2
...     PUSH = 3
...     PATCH = 4
...     DELETE = 5
...

>>> list(HTTPMethod)
[
    <HTTPMethod.GET: 1>,
    <HTTPMethod.POST: 2>,
    <HTTPMethod.PUSH: 3>,
    <HTTPMethod.PATCH: 4>,
    <HTTPMethod.DELETE: 5>
]
```

Here, you use the class syntax to define the `HTTPMethod` enum. This example is completely equivalent to the previous one, as you can conclude from the output of `list()`.

Using either the class syntax or the functional API to create your enumeration is your decision and will mostly depend on your taste and concrete conditions. However, if you want to create enumerations dynamically, then the functional API can be your only option.

Consider the following example, where you create an enum with user-provided members:

Python >>>

```
>>> from enum import Enum

>>> names = []
>>> while True:
...     name = input("Member name: ")
...     if name in {"q", "Q"}:
...         break
...     names.append(name.upper())
...
Member name: YES
Member name: NO
Member name: q

>>> DynamicEnum = Enum("DynamicEnum", names)
```

```
>>> list(DynamicEnum)
[<DynamicEnum.YES: 1>, <DynamicEnum.NO: 2>]
```

This example is a little bit extreme because creating any object from your user's input is quite a risky practice, considering that you can't predict what the user will input. However, the example is intended to show that the functional API is the way to go when you need to create enumerations dynamically.

Finally, if you need to set custom values for your enum members, then you can use an iterable of name-value pairs as your `names` argument. In the example below, you use a list of name-value tuples to initialize all the enumeration members:

Python >>>

```
>>> from enum import Enum

>>> HTTPStatusCode = Enum(
...     value="HTTPStatusCode",
...     names=[
...         ("OK", 200),
...         ("CREATED", 201),
...         ("BAD_REQUEST", 400),
...         ("NOT_FOUND", 404),
...         ("SERVER_ERROR", 500),
...     ],
... )

>>> list(HTTPStatusCode)
[
    <HTTPStatusCode.OK: 200>,
    <HTTPStatusCode.CREATED: 201>,
    <HTTPStatusCode.BAD_REQUEST: 400>,
    <HTTPStatusCode.NOT_FOUND: 404>,
    <HTTPStatusCode.SERVER_ERROR: 500>
]
```

Providing a list of name-value tuples like you did above makes it possible to create the `HTTPStatusCode` enumeration with custom values for the members. In this example, if you didn't want to use a list of name-value tuples, then you could also use a dictionary that maps names to values.

Building Enumerations From Automatic Values

Python's `enum` module provides a convenient function called `auto()` that allows you to set automatic values for your enum members. This function's default behavior is to assign consecutive integer values to members.

Here's how `auto()` works:

Python >>>

```
>>> from enum import auto, Enum

>>> class Day(Enum):
...     MONDAY = auto()
...     TUESDAY = auto()
...     WEDNESDAY = 3
...     THURSDAY = auto()
...     FRIDAY = auto()
...     SATURDAY = auto()
...     SUNDAY = 7
...

>>> list(Day)
[
    <Day.MONDAY: 1>,
    <Day.TUESDAY: 2>,
    <Day.WEDNESDAY: 3>,
    <Day.THURSDAY: 4>,
    <Day.FRIDAY: 5>,
    <Day.SATURDAY: 6>,
    <Day.SUNDAY: 7>
]
```

You need to call `auto()` once for each automatic value that you need. You can also combine `auto()` with concrete values, just like you did with `Day.WEDNESDAY` and `Day.SUNDAY` in this example.

By default, `auto()` assigns consecutive integer numbers to each target member starting from `1`. You can tweak this default behavior by overriding the `._generate_next_value_()` method, which `auto()` uses under the hood to generate the automatic values.

Here's an example of how to do this:

Python >>>

```
>>> from enum import Enum, auto

>>> class CardinalDirection(Enum):
...     def _generate_next_value_(name, start, count, last_valu
...         return name[0]
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...

>>> list(CardinalDirection)
[
    <CardinalDirection.NORTH: 'N'>,
    <CardinalDirection.SOUTH: 'S'>,
    <CardinalDirection.EAST: 'E'>,
    <CardinalDirection.WEST: 'W'>
]
```

In this example, you create an enumeration of Earth's cardinal directions in which values are automatically set to strings containing the first character of each member's name. Note that you must provide your overridden version of `._generate_next_value_()` before defining any members. That's because the members will be built by calling the method.

Creating Enumerations With Aliases and Unique Values

You can create enumerations in which two or more members have the same constant value. The redundant members are known as **aliases** and can be useful in some situations. For example, say that you have an enum containing a set of operating systems (OS), like in

the following code:

```
Python >>>
>>> from enum import Enum

>>> class OperatingSystem(Enum):
...     UBUNTU = "linux"
...     MACOS = "darwin"
...     WINDOWS = "win"
...     DEBIAN = "linux"
...

>>> # Aliases aren't listed
>>> list(OperatingSystem)
[
    <OperatingSystem.UBUNTU: 'linux'>,
    <OperatingSystem.MACOS: 'darwin'>,
    <OperatingSystem.WINDOWS: 'win'>
]

>>> # To access aliases, use __members__
>>> list(OperatingSystem.__members__.items())
[
    ('UBUNTU', <OperatingSystem.UBUNTU: 'linux'>),
    ('MACOS', <OperatingSystem.MACOS: 'darwin'>),
    ('WINDOWS', <OperatingSystem.WINDOWS: 'win'>),
    ('DEBIAN', <OperatingSystem.UBUNTU: 'linux'>)
]
```

Linux distributions are considered independent operating systems. So, Ubuntu and Debian are both independent systems with different goals and target audiences. However, they share a common kernel called Linux.

The above enumeration maps operating systems to their corresponding kernels. This relationship turns `DEBIAN` into an alias of `UBUNTU`, which may be useful when you have code that's kernel-related along with code that's specific to a given Linux distribution.

An important piece of behavior to note in the above example is that when you iterate over the enumeration directly, aliases aren't considered. If you ever need to iterate over all the members, including aliases, then you need to use `.__members__`. You'll learn

more about iteration and the `.__members__` attribute in the section about iterating through enumerations.

You also have the option to completely forbid aliases in your enumerations. To do this, you can use the `@unique` decorator from the `enum` module:

```
Python >>>

>>> from enum import Enum, unique

>>> @unique
... class OperatingSystem(Enum):
...     UBUNTU = "linux"
...     MACOS = "darwin"
...     WINDOWS = "win"
...     DEBIAN = "linux"
...
Traceback (most recent call last):
    ...
ValueError: duplicate values in <enum 'OperatingSystem'>: DEBIA
```

In this example, you decorate `OperatingSystem` with `@unique`. If any member value is duplicated, then you get a `ValueError`. Here, the exception message points out that `DEBIAN` and `UBUNTU` share the same value, which isn't allowed.

Working With Enumerations in Python

Up to this point, you've learned what enumerations are, when to use them, and what benefits you get from using them in your code. You've also leaned how to create enumerations in Python using the `Enum` class either as a superclass or as a callable.

Now it's time for you to start digging into how Python's enumerations work and how you can use them in your code.

Accessing Enumeration Members

When it comes to using enumerations in your code, accessing their members is a fundamental operation to perform. You'll have three different ways to access enumeration members in Python.

For example, say that you need to access the `NORTH` member of the `CardinalDirection` enum below. In this situation, you can do something like this:

Python >>>

```
>>> from enum import Enum

>>> class CardinalDirection(Enum):
...     NORTH = "N"
...     SOUTH = "S"
...     EAST = "E"
...     WEST = "W"
...

>>> # Dot notation
>>> CardinalDirection.NORTH
<CardinalDirection.NORTH: 'N'>

>>> # Call notation
>>> CardinalDirection("N")
<CardinalDirection.NORTH: 'N'>

>>> # Subscript notation
>>> CardinalDirection["NORTH"]
<CardinalDirection.NORTH: 'N'>
```

The first highlighted line in this example shows how you can access an enum member using the **dot notation**, which is pretty intuitive and readable. The second highlighted line accesses the target member by **calling** the enumeration with the member's value as an argument.

> **Note:** It's important to note that calling an enumeration with a member's value as an argument can make you feel like you're instantiating the enumeration. However, enumerations can't

be instantiated, as you already know:

Python >>>

```
>>> week = Day()
Traceback (most recent call last):
    ...
TypeError: EnumMeta.__call__() missing 1 required positio
```

Trying to create an instance of an existing enumeration isn't allowed, so you get a `TypeError` if you attempt to do it. Therefore, you must not confuse instantiating with accessing members through an enumeration call.

Finally, the third highlighted line shows how you can use a **dictionary-like notation** or **subscript notation** to access a member using the member's name as the target key.

Python's enumerations offer great flexibility for you to access members. The dot notation is arguably the most commonly used approach in Python code. However, the other two approaches can be helpful as well. So, use the notation that fulfills your specific needs, conventions, and style.

Using the `.name` and `.value` Attributes

The members of a Python enumeration are instances of their containing class. During the enum class parsing, each member is automatically provided with a `.name` attribute that holds the member's name as a string. Members also get a `.value` attribute that stores the value assigned to the member itself in the class definition.

You can access `.name` and `.value` as you'd do with a regular attribute, using the dot notation. Consider the following example, which simulates a semaphore, more commonly known as a traffic light:

Python >>>

```
>>> from enum import Enum
```

```
>>> class Semaphore(Enum):
...     RED = 1
...     YELLOW = 2
...     GREEN = 3
...

>>> Semaphore.RED.name
'RED'

>>> Semaphore.RED.value
1

>>> Semaphore.YELLOW.name
'YELLOW'
```

The `.name` and `.value` attributes of an enum member give you direct access to the member's name as a string and to the member's value, respectively. These attributes come in handy when you're iterating through your enumerations, which you'll explore in the next section.

Iterating Through Enumerations

A remarkable feature of Python enumerations compared to regular classes is that enumerations are iterable by default. Because they're iterable, you can use them in `for` loops and with other tools that accept and process iterables.

Python's enumerations support direct iteration over members in the definition order:

Python >>>

```
>>> from enum import Enum

>>> class Flavor(Enum):
...     VANILLA = 1
...     CHOCOLATE = 2
...     MINT = 3
...

>>> for flavor in Flavor:
...     print(flavor)
...
Flavor.VANILLA
```

```
Flavor.CHOCOLATE
Flavor.MINT
```

In this example, you use a `for` loop to iterate over the members of `Flavor`. Note that members are produced in the same order as they were defined in the class definition.

When you're iterating over an enumeration, you can access the `.name` and `.value` attributes as you go:

Python >>>

```
>>> for flavor in Flavor:
...     print(flavor.name, "->", flavor.value)
...
VANILLA -> 1
CHOCOLATE -> 2
MINT -> 3
```

This kind of iteration technique looks pretty similar to iterating over a dictionary. So, if you're familiar with dictionary iteration, then looping over enumerations using this technique will be a straightforward task with many potential use cases.

Alternatively, enumerations have a special attribute called `.__members__` that you can also use for iterating over their members. This attribute holds a dictionary that maps names to members. The difference between iterating over this dictionary and over the enumeration directly is that the dictionary gives you access to all members of the enumeration, including all the aliases that you may have.

Here are some examples of using `.__members__` to iterate through your `Flavor` enumeration:

Python >>>

```
>>> for name in Flavor.__members__:
...     print(name)
...
VANILLA
CHOCOLATE
MINT
```

```
>>> for name in Flavor.__members__.keys():
...     print(name)
...
VANILLA
CHOCOLATE
MINT

>>> for member in Flavor.__members__.values():
...     print(member)
...
Flavor.VANILLA
Flavor.CHOCOLATE
Flavor.MINT

>>> for name, member in Flavor.__members__.items():
...     print(name, "->", member)
...
VANILLA -> Flavor.VANILLA
CHOCOLATE -> Flavor.CHOCOLATE
MINT -> Flavor.MINT
```

You can use the `.__members__` special attribute for detailed programmatic access to the members of a Python enumeration. Because `.__members__` holds a regular dictionary, you can use all the iteration techniques that apply to this built-in data type. Some of these techniques include using dictionary methods like `.key()`, `.values()`, and `.items()`.

Using Enumerations in `if` and `match` Statements

Chained `if ... elif` statements and the relatively new `match ... case` statement are common and arguably natural places where you can use enumerations. Both constructs allow you to take different courses of action depending on certain conditions.

For example, say that you have a piece of code that handles a

semaphore, or traffic light, in a traffic control application. You must perform different actions depending on the current light of the semaphore. In this situation, you can use an enumeration to represent the semaphore and its lights. Then you can use a chain of `if` … `elif` statements to decide on the action to run:

Python >>>

```
>>> from enum import Enum

>>> class Semaphore(Enum):
...     RED = 1
...     YELLOW = 2
...     GREEN = 3
...

>>> def handle_semaphore(light):
...     if light is Semaphore.RED:
...         print("You must stop!")
...     elif light is Semaphore.YELLOW:
...         print("Light will change to red, be careful!")
...     elif light is Semaphore.GREEN:
...         print("You can continue!")
...

>>> handle_semaphore(Semaphore.GREEN)
You can continue!

>>> handle_semaphore(Semaphore.YELLOW)
Light will change to red, be careful!

>>> handle_semaphore(Semaphore.RED)
You must stop!
```

The chain of `if` … `elif` statements in your `handle_semaphore()` function checks the value of the current light to decide on the action to take. Note that the calls to `print()` in `handle_semaphore()` are just placeholders. In real code, you'd replace them with more complex operations.

If you're using Python 3.10 or greater, then you can quickly turn the above chain of `if` … `elif` statements into an equivalent `match` … `case` statement:

Python >>>

```
>>> from enum import Enum

>>> class Semaphore(Enum):
...     RED = 1
...     YELLOW = 2
...     GREEN = 3
...

>>> def handle_semaphore(light):
...     match light:
...         case Semaphore.RED:
...             print("You must stop!")
...         case Semaphore.YELLOW:
...             print("Light will change to red, be careful!")
...         case Semaphore.GREEN:
...             print("You can continue!")
...

>>> handle_semaphore(Semaphore.GREEN)
You can continue!

>>> handle_semaphore(Semaphore.YELLOW)
Light will change to red, be careful!

>>> handle_semaphore(Semaphore.RED)
You must stop!
```

This new implementation of `handle_semaphore()` is equivalent to the previous implementation that uses `if` … `elif` statements. Using either technique is a matter of taste and style. Both techniques work well and are comparable in terms of readability. However, note that if you need to guarantee backward compatibility with Python versions lower than 3.10, then you must use chained `if` … `elif` statements.

Finally, note that even though enumerations seem to play well with `if` … `elif` and `match` … `case` statements, you must keep in mind that these statements don't scale well. If you add new members to your target enumeration, then you'll need to update the handling function to consider these new members.

Comparing Enumerations

Being able to use enumerations in `if … elif` statements and `match … case` statements suggests that enumeration members can be compared. By default, enums support two types of comparison operators:

1. **Identity**, using the `is` and `is not` operators
2. **Equality**, using the `==` and `!=` operators

The identity comparison relies on the fact that each enum member is a singleton instance of its enumeration class. This characteristic allows for fast and cheap identity comparison of members using the `is` and `is not` operators.

Consider the following examples, which compare different combinations of enum members:

Python >>>

```
>>> from enum import Enum

>>> class AtlanticAveSemaphore(Enum):
...     RED = 1
...     YELLOW = 2
...     GREEN = 3
...     PEDESTRIAN_RED = 1
...     PEDESTRIAN_GREEN = 3
...

>>> red = AtlanticAveSemaphore.RED
>>> red is AtlanticAveSemaphore.RED
True
>>> red is not AtlanticAveSemaphore.RED
False

>>> yellow = AtlanticAveSemaphore.YELLOW
>>> yellow is red
False
>>> yellow is not red
True

>>> pedestrian_red = AtlanticAveSemaphore.PEDESTRIAN_RED
>>> red is pedestrian_red
True
```

Every enum member has its own identity, which is different from the identity of its sibling members. This rule doesn't apply to member aliases, because they're just references to existing members and share the same identity. This is why comparing `red` and `pedestrian_red` returns `True` in your final example.

> **Note:** To get the identity of a given object in Python, you can use the built-in `id()` function with the object as an argument.

Identity checks between members of different enumerations always return `False`:

Python >>>

```
>>> class EighthAveSemaphore(Enum):
...     RED = 1
...     YELLOW = 2
...     GREEN = 3
...     PEDESTRIAN_RED = 1
...     PEDESTRIAN_GREEN = 3
...

>>> AtlanticAveSemaphore.RED is EighthAveSemaphore.RED
False

>>> AtlanticAveSemaphore.YELLOW is EighthAveSemaphore.YELLOW
False
```

The reason for this falsy result is that members of different enums are independent instances with their own identities, so any identity check on them returns `False`.

The equality operators `==` and `!=` also work between enumeration members:

Python >>>

```
>>> from enum import Enum

>>> class AtlanticAveSemaphore(Enum):
...     RED = 1
...     YELLOW = 2
```

```
...     GREEN = 3
...     PEDESTRIAN_RED = 1
...     PEDESTRIAN_GREEN = 3
...

>>> red = AtlanticAveSemaphore.RED
>>> red == AtlanticAveSemaphore.RED
True

>>> red != AtlanticAveSemaphore.RED
False

>>> yellow = AtlanticAveSemaphore.YELLOW
>>> yellow == red
False
>>> yellow != red
True

>>> pedestrian_red = AtlanticAveSemaphore.PEDESTRIAN_RED
>>> red == pedestrian_red
True
```

Python's enumerations support both operators, `==` and `!=`, by delegating to the `is` and `is not` operators, respectively.

As you already learned, enum members always have a concrete value that can be a number, a string, or any other object. Because of this, running equality comparisons between enum members and common objects can be tempting.

However, this kind of comparison doesn't work as expected because the actual comparison is based on object identity:

Python >>>

```
>>> from enum import Enum

>>> class Semaphore(Enum):
...     RED = 1
...     YELLOW = 2
...     GREEN = 3
...

>>> Semaphore.RED == 1
False
```

```
>>> Semaphore.YELLOW == 2
False

>>> Semaphore.GREEN != 3
True
```

Even though the member values are equal to the integers in each example, these comparisons return `False`. This is because regular enum members compare by object identity rather than by value. In the example above, you're comparing enum members to integer numbers, which is like comparing apples and oranges. They'll never compare equally, because they have different identities.

> **Note:** Later, you'll learn about `IntEnum` which are special enumerations that can be compared to integers.

Finally, another comparison-related feature of enumerations is that you can perform membership tests on them using the `in` and `not in` operators:

Python >>>

```
>>> from enum import Enum

>>> class Semaphore(Enum):
...     RED = 1
...     YELLOW = 2
...     GREEN = 3
...

>>> Semaphore.RED in Semaphore
True

>>> Semaphore.GREEN not in Semaphore
False
```

Python's enumerations support the `in` and `not in` operators by default. Using these operators, you can check if a given member is present in a given enumeration.

Sorting Enumerations

By default, Python's enums don't support comparison operators like >, <, >=, and <=. That's why you can't sort the members of an enumeration using the built-in `sorted()` function directly, like in the example below:

Python >>>

```
>>> from enum import Enum

>>> class Season(Enum):
...     SPRING = 1
...     SUMMER = 2
...     AUTUMN = 3
...     WINTER = 4
...

>>> sorted(Season)
Traceback (most recent call last):
    ...
TypeError: '<' not supported between instances of 'Season' and
```

When you use an enumeration as an argument to `sorted()`, you get a `TypeError` because enums don't support the `<` operator. However, there's a way to successfully sort enumerations by their members' names and values using the `key` argument in the `sorted()` call.

Here's how to do it:

Python >>>

```
>>> sorted(Season, key=lambda season: season.value)
[
    <Season.SPRING: 1>,
    <Season.SUMMER: 2>,
    <Season.AUTUMN: 3>,
    <Season.WINTER: 4>
]

>>> sorted(Season, key=lambda season: season.name)
[
```

```
    <Season.AUTUMN: 3>,
    <Season.SPRING: 1>,
    <Season.SUMMER: 2>,
    <Season.WINTER: 4>
]
```

In the first example, you use a `lambda` function that takes an enumeration member as an argument and returns its `.value` attribute. With this technique, you can sort the input enumeration by its values. In the second example, the `lambda` function takes an enum member and returns its `.name` attribute. This way, you can sort the enumeration by the names of its members.

Extending Enumerations With New Behavior

In the previous sections, you've learned how to create and use enumerations in your Python code. Up to this point, you've worked with default enumerations. This means that you've used Python's enumerations with their standard features and behaviors only.

Sometimes, you may need to provide your enumerations with custom behavior. To do this, you can add methods to your enums and implement the required functionality. You can also use mixin classes. In the following sections, you'll learn how to take advantage of both techniques to customize your enumerations.

Adding and Tweaking Member Methods

You can provide your enumerations with new functionality by adding new methods to your enumeration classes as you'd do with any regular Python class. Enumerations are classes with special features. Like regular classes, enumerations can have methods and special methods.

Consider the following example, adapted from the Python documentation:

```
Python                                                    >>>
```

```
>>> from enum import Enum

>>> class Mood(Enum):
...     FUNKY = 1
...     MAD = 2
...     HAPPY = 3
...
...     def describe_mood(self):
...         return self.name, self.value
...
...     def __str__(self):
...         return f"I feel {self.name}"
...
...     @classmethod
...     def favorite_mood(cls):
...         return cls.HAPPY
...

>>> Mood.HAPPY.describe_mood()
('HAPPY', 3)

>>> print(Mood.HAPPY)
I feel HAPPY

>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
```

In this example, you have a `Mood` enumeration with three members. Regular methods like `.describe_mood()` are bound to instances of their containing enum, which are the enum members. So, you must call regular methods on enum members rather than on the enum class itself.

> **Note:** Remember that Python's enumerations can't be instantiated. The members of an enumeration are the enumeration's allowed instances. So, the `self` parameter represents the current member.

Similarly, the `.__str__()` special method operates on members, providing a nicely printable representation of each member.

Finally, the `.favorite_mood()` method is a class method, which

operates on the class or enumeration itself. Class methods like this one provide access to all the enum members from inside the class.

You can also take advantage of this ability to contain additional behavior when you need to implement the strategy pattern. For example, say you need a class that allows you to use two strategies for sorting a list of numbers in ascending and descending order. In this case, you can use an enumeration like the following:

Python >>>

```
>>> from enum import Enum

>>> class Sort(Enum):
...     ASCENDING = 1
...     DESCENDING = 2
...     def __call__(self, values):
...         return sorted(values, reverse=self is Sort.DESCENDI
...

>>> numbers = [5, 2, 7, 6, 3, 9, 8, 4]

>>> Sort.ASCENDING(numbers)
[2, 3, 4, 5, 6, 7, 8, 9]

>>> Sort.DESCENDING(numbers)
[9, 8, 7, 6, 5, 4, 3, 2]
```

Each member of `Sort` represents a sorting strategy. The `.__call__()` method makes the members of `Sort` callable. Inside `.__call__()`, you use the built-in `sorted()` function to sort the input values in ascending or descending order, depending on the called member.

> **Note:** The above example is intended to be a demonstrative example of using an enum to implement the strategy design pattern. In practice, it's unnecessary to create this `Sort` enum with the sole purpose of wrapping the `sorted()` function. Instead, you may use `sorted()` and its `reverse` argument directly and avoid overengineering your solution.

When you call `Sort.ASCENDING`, the input numbers are sorted in

ascending order. In contrast, when you call `Sort.DESCENDING`, the numbers get sorted in descending order. That's it! You've used an enumeration to quickly implement the strategy design pattern.

Mixing Enumerations With Other Types

Python supports multiple inheritance as part of its object-oriented features. This means that in Python, you can inherit multiple classes when creating class hierarchies. Multiple inheritance comes in handy when you want to reuse functionality from several classes at the same time.

A common practice in object-oriented programming is to use what's known as mixin classes. These classes provide functionality that other classes can use. In Python, you can add mixin classes to the list of parents of a given class to automatically get the mixin functionality.

For example, say that you want an enumeration that supports integer comparison. In this case, you can use the built-in `int` type as a mixin when defining your enum:

Python >>>

```
>>> from enum import Enum

>>> class Size(int, Enum):
...     S = 1
...     M = 2
...     L = 3
...     XL = 4
...

>>> Size.S > Size.M
False
>>> Size.S < Size.M
True
>>> Size.L >= Size.M
```

```
True
>>> Size.L <= Size.M
False

>>> Size.L > 2
True
>>> Size.M < 1
False
```

In this example, your `Size` class inherits from `int` and `Enum`. Inheriting from the `int` type enables direct comparison between members through the `>`, `<`, `>=`, and `<=` comparison operators. It also enables comparisons between `Size` members and integer numbers.

Finally, note that when you use a data type as a mixin, the member's `.value` attribute isn't the same as the member itself, although it's equivalent and will compare as such. That's why you can compare the members of `Size` with integer numbers directly.

> **Note:** Using integer enum member values is a pretty common practice. That's why the `enum` module provides an `IntEnum` to create enumerations with integer values directly. You'll learn more about this class in the section called Exploring Other Enumeration Classes.

The above example shows that creating enumerations with mixin classes is often of great help when you need to reuse a given piece of functionality. If you decide to use this technique in some of your enums, then you'll have to stick to the following signature:

Python

```
class EnumName([mixin_type, ...], [data_type,] enum_type):
    # Members go here...
```

This signature implies that you can have one or more mixin classes, at most one data type class, and the parent enum class, in that order.

Consider the following examples:

Python >>>

```
>>> from enum import Enum

>>> class MixinA:
...     def a(self):
...         print(f"MixinA: {self.value}")
...

>>> class MixinB:
...     def b(self):
...         print(f"MixinB: {self.value}")
...

>>> class ValidEnum(MixinA, MixinB, str, Enum):
...     MEMBER = "value"
...

>>> ValidEnum.MEMBER.a()  # Call .a() from MixinA
MixinA: value

>>> ValidEnum.MEMBER.b()  # Call .b() from MixinB
MixinB: value

>>> ValidEnum.MEMBER.upper()  # Call .upper() from str
'VALUE'

>>> class WrongMixinOrderEnum(Enum, MixinA, MixinB):
...     MEMBER = "value"
...
Traceback (most recent call last):
    ...
TypeError: new enumerations should be created as
    `EnumName([mixin_type, ...] [data_type,] enum_type)`

>>> class TooManyDataTypesEnum(int, str, Enum):
...     MEMBER = "value"
...
Traceback (most recent call last):
    ...
TypeError: 'TooManyDataTypesEnum': too many data types:
    {<class 'int'>, <class 'str'>}
```

The `ValidEnum` class shows that in the sequence of bases, you must place as many mixin classes as you need—but only one data type—before `Enum`.

`WrongMixinOrderEnum` shows that if you put `Enum` in any position other than the last, then you'll get a `TypeError` with information about the correct signature to use. Meanwhile, `TooManyDataTypesEnum` confirms that your list of mixin classes must have at most one concrete data type, such as `int` or `str`.

Keep in mind that if you use a concrete data type in your list of mixin classes, then the member values have to match the type of this specific data type.

Exploring Other Enumeration Classes

Apart from `Enum`, the `enum` module provides a few additional classes that allow you to create enumerations with specific behaviors. You'll have the `IntEnum` class for creating enumerated constants that are also subclasses of `int`, which implies that all members will have all the features of an integer number.

You'll also find more specialized classes, like `IntFlag` and `Flag`. Both classes will allow you to create enumerated sets of constants that you can combine using the bitwise operators. In the following section, you'll explore these classes and how they work in Python.

Building Integer Enumerations: `IntEnum`

Integer enumerations are so common that the `enum` module exports a dedicated class called `IntEnum` that was specifically created to cover this use case. If you need the members of your enumerations to behave like integer numbers, then you should inherit from `IntEnum` rather than from `Enum`.

Subclassing `IntEnum` is equivalent to using multiple inheritance with `int` as the mixin class:

Python >>>

```
>>> from enum import IntEnum

>>> class Size(IntEnum):
```

```
...     S = 1
...     M = 2
...     L = 3
...     XL = 4
...

>>> Size.S > Size.M
False
>>> Size.S < Size.M
True
>>> Size.L >= Size.M
True
>>> Size.L <= Size.M
False

>>> Size.L > 2
True
>>> Size.M < 1
False
```

Now `Size` inherits directly from `IntEnum` instead of from `int` and `Enum`. Like the previous version of `Size`, this new version has full comparison capabilities and supports all the comparison operators. You can also use the class members in integer operations directly.

`Size` will automatically attempt to convert any value of a different data type to an integer number. If this conversion isn't possible, then you'll get a `ValueError`:

Python >>>

```
>>> from enum import IntEnum

>>> class Size(IntEnum):
...     S = 1
...     M = 2
...     L = 3
...     XL = "4"
...

>>> list(Size)
[<Size.S: 1>, <Size.M: 2>, <Size.L: 3>, <Size.XL: 4>]

>>> class Size(IntEnum):
...     S = 1
...     M = 2
```

```
...     L = 3
...     XL = "4.o"
...
Traceback (most recent call last):
    ...
ValueError: invalid literal for int() with base 10: '4.o'
```

In the first example, `Size` automatically converts the string `"4"` into an integer value. In the second example, because the string `"4.o"` doesn't hold a valid numeric value, you get a `ValueError`, and the conversion fails.

In the current stable Python version, 3.10, the `enum` module doesn't include a `StrEnum` class. However, this class is another example of a popular use case of enumerations. For this reason, Python 3.11 will include a `StrEnum` type with direct support for common string operations. In the meantime, you can simulate the behavior of a `StrEnum` class by creating a mixin class with `str` and `Enum` as parent classes.

Creating Integer Flags: `IntFlag` and `Flag`

You can use `IntFlag` as the base class for enumerations that should support the bitwise operators. Performing bitwise operations on members of an `IntFlag` subclass will return an object that's also a member of the underlying enum.

Here's an example of a `Role` enumeration that lets you manage different user roles in a single combined object:

Python >>>

```
>>> from enum import IntFlag

>>> class Role(IntFlag):
...     OWNER = 8
...     POWER_USER = 4
```

```
...     USER = 2
...     SUPERVISOR = 1
...     ADMIN = OWNER | POWER_USER | USER | SUPERVISOR
...

>>> john_roles = Role.USER | Role.SUPERVISOR
>>> john_roles
<Role.USER|SUPERVISOR: 3>

>>> type(john_roles)
<enum 'Role'>

>>> if Role.USER in john_roles:
...     print("John, you're a user")
...
John, you're a user

>>> if Role.SUPERVISOR in john_roles:
...     print("John, you're a supervisor")
...
John, you're a supervisor

>>> Role.OWNER in Role.ADMIN
True

>>> Role.SUPERVISOR in Role.ADMIN
True
```

In this code snippet, you create an enumeration that holds a set of user roles in a given application. The members of this enumeration hold integer values that you can combine using the bitwise OR operator (`|`). For example, the user named John has both `USER` and `SUPERVISOR` roles. Note that the object stored in `john_roles` is a member of your `Role` enumeration.

> **Note:** You should keep in mind that individual members of enums based on `IntFlag`, also known as **flags**, should take values that are powers of two (1, 2, 4, 8, …). However, this isn't a requirement for combinations of flags, like `Role.ADMIN`.

In the above example, you defined `Role.ADMIN` as a combination of roles. Its value results from applying the bitwise OR operator to the complete list of previous roles in the enumeration.

`IntFlag` also supports integer operations, such as arithmetic and comparison operations. However, these types of operations return integers rather than member objects:

Python >>>

```
>>> Role.ADMIN + 1
16

>>> Role.ADMIN - 2
13

>>> Role.ADMIN / 3
5.0

>>> Role.ADMIN < 20
True
```

`IntFlag` members are also subclasses of `int`. That's why you can use them in expressions that involve integer numbers. In these situations, the resulting value will be an integer rather than an enum member.

Finally, you'll also find the `Flag` class available in `enum`. This class works similarly to `IntFlag` and has some additional restrictions:

Python >>>

```
>>> from enum import Flag

>>> class Role(Flag):
...     OWNER = 8
...     POWER_USER = 4
...     USER = 2
...     SUPERVISOR = 1
...     ADMIN = OWNER | POWER_USER | USER | SUPERVISOR
...

>>> john_roles = Role.USER | Role.SUPERVISOR
>>> john_roles
<Role.USER|SUPERVISOR: 3>

>>> type(john_roles)
<enum 'Role'>
```

```
>>> if Role.USER in john_roles:
...     print("John, you're a user")
...
John, you're a user

>>> if Role.SUPERVISOR in john_roles:
...     print("John, you're a supervisor")
...
John, you're a supervisor

>>> Role.OWNER in Role.ADMIN
True

>>> Role.SUPERVISOR in Role.ADMIN
True

>>> Role.ADMIN + 1
Traceback (most recent call last):
    ...
TypeError: unsupported operand type(s) for +: 'Role' and 'int'
```

The main difference between `IntFlag` and `Flag` is that the latter doesn't inherit from `int`. Therefore, integer operations aren't supported. When you try to use a member of `Role` in an integer operation, you get a `TypeError`.

Just like members of `IntFlag` enums, the members of `Flag` enums should have values that are powers of two. Again, this doesn't apply to combinations of flags, like `Role.ADMIN` in the example above.

Using Enumerations: Two Practical Examples

Python's enumerations can help you improve your code's readability and organization. You can use them to group similar constants that you can then use in your code to replace strings, numbers, and other values with readable and meaningful names.

In the following sections, you'll code a couple of practical examples that deal with common enum use cases. These examples will help you decide when your code could benefit from using enumerations.

Replacing Magic Numbers

Enumerations are great when you need to replace sets of related magic numbers, such as HTTP status codes, computer ports, and exit codes. With an enumeration, you can group these numeric constants and assign them readable and descriptive names that you can use and reuse in your code later.

Say that you have the following function as part of an application that retrieves and processes HTTP content directly from the web:

Python >>>

```
>>> from http.client import HTTPSConnection

>>> def process_response(response):
...     match response.getcode():
...         case 200:
...             print("Success!")
...         case 201:
...             print("Successfully created!")
...         case 400:
...             print("Bad request")
...         case 404:
...             print("Not Found")
...         case 500:
...             print("Internal server error")
...         case _:
...             print("Unexpected status")
...

>>> connection = HTTPSConnection("www.python.org")
>>> try:
...     connection.request("GET", "/")
...     response = connection.getresponse()
...     process_response(response)
... finally:
...     connection.close()
...
Success!
```

Your `process_response()` function takes an HTTP `response` object as an argument. Then it gets the status code from `response` using the `.getcode()` method. The `match ... case` statement sequentially

compares the current status code with some standard status codes provided as magic numbers in your example.

If a match occurs, then the code block in the matched `case` runs. If no match occurs, then the default `case` runs. Note that the default `case` is the one that uses an underscore (_) as a match criterion.

The rest of the code connects to a sample web page, performs a `GET` request, retrieves the response object, and processes it using your `process_response()` function. The `finally` clause closes the active connection to avoid resource leaks.

Even though this code works, it can be challenging to read and understand for people unfamiliar with HTTP status codes and their corresponding meanings. To fix these issues and make your code more readable and maintainable, you can use an enumeration to group the HTTP status codes and give them descriptive names:

Python >>>

```
>>> from enum import IntEnum
>>> from http.client import HTTPSConnection

>>> class HTTPStatusCode(IntEnum):
...     OK = 200
...     CREATED = 201
...     BAD_REQUEST = 400
...     NOT_FOUND = 404
...     SERVER_ERROR = 500
...

>>> def process_response(response):
...     match response.getcode():
...         case HTTPStatusCode.OK:
...             print("Success!")
...         case HTTPStatusCode.CREATED:
...             print("Successfully created!")
...         case HTTPStatusCode.BAD_REQUEST:
...             print("Bad request")
...         case HTTPStatusCode.NOT_FOUND:
...             print("Not Found")
...         case HTTPStatusCode.SERVER_ERROR:
...             print("Internal server error")
...         case _:
...             print("Unexpected status")
```

```
...

>>> connection = HTTPSConnection("www.python.org")
>>> try:
...     connection.request("GET", "/")
...     response = connection.getresponse()
...     process_response(response)
... finally:
...     connection.close()
...
Success!
```

This code adds a new enumeration called `HTTPStatusCode` to your application. This enum groups the target HTTP status codes and gives them readable names. It also makes them strictly constant, which makes your app more reliable.

Inside `process_response()`, you use human-readable and descriptive names that provide context and content information. Now anyone reading your code will immediately know that the match criteria are HTTP status codes. They'll also spot the meaning of each target code quickly.

Creating a State Machine

Another interesting use case of enumerations is when you use them for re-creating the different possible states of a given system. If your system can be in exactly one of a finite number of states at any given time, then your system works as a state machine. Enumerations are useful when you need to implement this common design pattern.

As an example of how to use an enum to implement the state machine pattern, you create a minimal disk player simulator. To start, go ahead and create a `disk_player.py` file with the following content:

Python

```
# disk_player.py

from enum import Enum, auto

class State(Enum):
    EMPTY = auto()
    STOPPED = auto()
    PAUSED = auto()
    PLAYING = auto()
```

Here, you define the `State` class. This class groups all the possible states of your disk player: `EMPTY`, `STOPPED`, `PAUSED`, and `PLAYING`. Now you can code the `DiskPlayer` player class, which would look something like this:

Python

```
# disk_player.py
# ...

class DiskPlayer:
    def __init__(self):
        self.state = State.EMPTY

    def insert_disk(self):
        if self.state is State.EMPTY:
            self.state = State.STOPPED
        else:
            raise ValueError("disk already inserted")

    def eject_disk(self):
        if self.state is State.EMPTY:
            raise ValueError("no disk inserted")
        else:
            self.state = State.EMPTY

    def play(self):
        if self.state in {State.STOPPED, State.PAUSED}:
            self.state = State.PLAYING

    def pause(self):
        if self.state is State.PLAYING:
            self.state = State.PAUSED
        else:
            raise ValueError("can't pause when not playing")
```

```
    def stop(self):
        if self.state in {State.PLAYING, State.PAUSED}:
            self.state = State.STOPPED
        else:
            raise ValueError("can't stop when not playing or pa
```

The `DiskPlayer` class implements all the possible actions that your player can execute, including inserting and ejecting the disk, playing, pausing, and stopping the player. Note how each method in `DiskPlayer` checks and updates the player's current state by taking advantage of your `State` enumeration.

To complete your example, you'll use the traditional `if __name__ == "__main__":` idiom to wrap up a few lines of code that'll allow you to try out the `DiskPlayer` class:

Python

```
# disk_player.py
# ...

if __name__ == "__main__":
    actions = [
        DiskPlayer.insert_disk,
        DiskPlayer.play,
        DiskPlayer.pause,
        DiskPlayer.stop,
        DiskPlayer.eject_disk,
        DiskPlayer.insert_disk,
        DiskPlayer.play,
        DiskPlayer.stop,
        DiskPlayer.eject_disk,
    ]
    player = DiskPlayer()
    for action in actions:
        action(player)
        print(player.state)
```

In this code snippet, you first define an `actions` variable, which holds the sequence of methods that you'll call from `DiskPlayer` in order to try out the class. Then you create an instance of your disk player class. Finally, you start a `for` loop to iterate over the list of actions and run every action through the `player` instance.

That's it! Your disk player simulator is ready for a test. To run it, go ahead and execute the following command at your command line:

Shell

```
$ python disk_player.py
State.STOPPED
State.PLAYING
State.PAUSED
State.STOPPED
State.EMPTY
State.STOPPED
State.PLAYING
State.STOPPED
State.EMPTY
```

This command's output shows that your app has gone through all the possible states. Of course, this example is minimal and doesn't consider all the potential scenarios. It's a demonstrative example of how you could use an enumeration to implement the state machine pattern in your code.

Conclusion

You now know how to create and use **enumerations** in Python. Enumerations, or just **enums**, are common and popular data types in many programming languages. With enumerations, you can group sets of related constants and access them through the enumeration itself.

Python doesn't provide a dedicated enum syntax. However, the `enum` module supports this common data type through the `Enum` class.

In this tutorial, you've learned how to:

- Create your own **enumerations** using Python's `Enum` class
- Work with enumerations and their **members**
- Provide your enumeration classes with **new functionalities**
- Use enumerations with some **practical examples**

You also learned about other useful enumeration types, such as

`IntEnum`, `IntFlag`, and `Flag`. They're available in `enum` and will help you create specialized enums.

With all this knowledge, you're now ready to start using Python's enums to group, name, and handle sets of semantically related constants. Enumerations allow you to better organize your code, making it more readable, explicit, and maintainable.

Source Code: Click here to download the free source code that you'll use to build enumerations in Python.

About **Leodanis Pozo Ramos**

Leodanis is an industrial engineer who loves Python and software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

» More about Leodanis

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 Bartosz

 Geir Arne

 Kate

Level Up Your Python Skills »

What Do You Think?

Rate this article:

Tweet Share Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

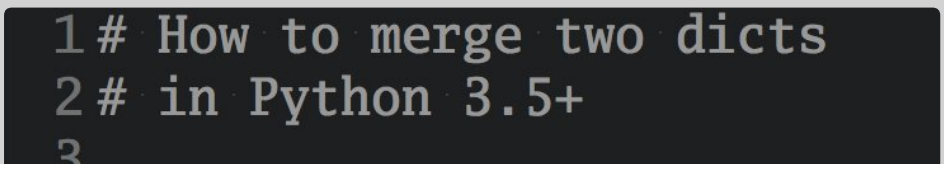
Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: intermediate python

— FREE Email Series —

Python Tricks



```
>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email…

Get Python Tricks »

No spam. Unsubscribe any time.

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Master Real-World Python Skills With a Community of Experts

Level Up With Unlimited Access to Our Vast Library of Python Tutorials and Video Lessons

Table of Contents