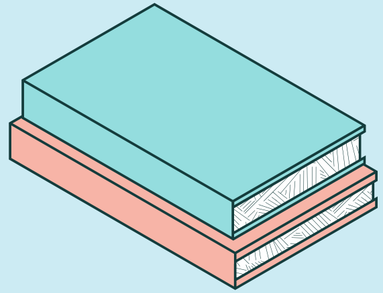
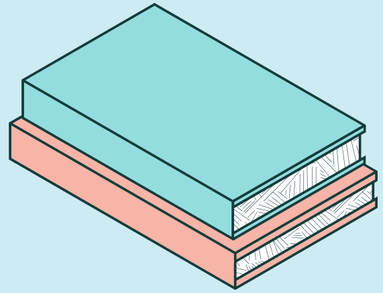
recap



- enumerations will primarily help us solve the problem of managing a static collection of fixed values

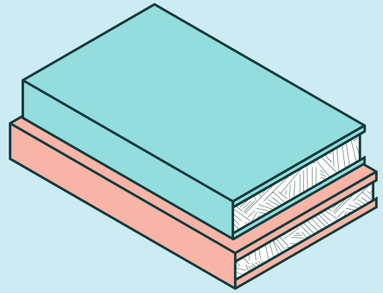
- they are a well-supported construct across many popular OOP languages like C++, C# and Java

recap



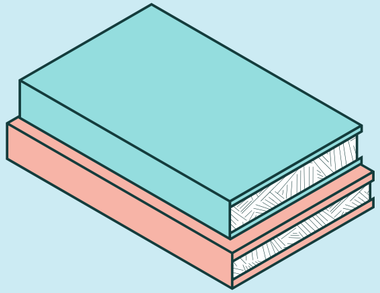
- we define enums as classes that inherit from enum.Enum
- they're an excellent choice for managing global variable-like constructs the OOP way
- more specifically they help associate various symbols to values along a single dimension/variable
- enums are by default immutable, iterable, and hashable

recap



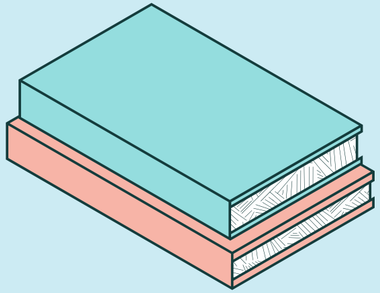
- each symbolic name <-> value association defined in the enum is known as a member
- members are instances of the enum type we define
- they are accessible by their symbolic name and also by the value that name points to

recap



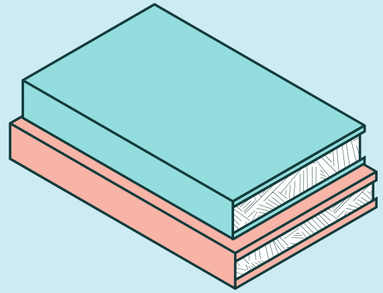
- enums must have unique symbolic names
- but more than one name could point to the same value
- when this happens, the first member defined for a given value is known as the master, and all that come after as its aliases
- looking up by value, by master name, or alias name always returns the master enum member

recap



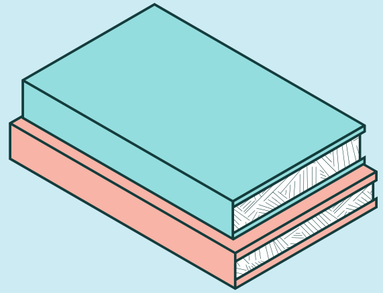
- in python enums, symbolic names should be unique, but values need not
- to enforce uniqueness over values too, we could use the unique() decorator from the enum module

recap



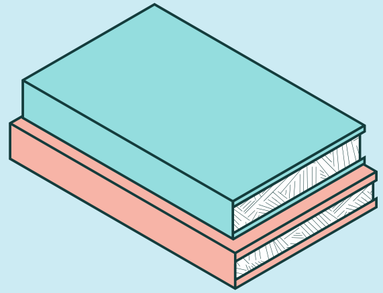
- when defining enums we need to associate the symbolic names with certain values
- in some domains and applications, the values themselves do not matter
- when that's the case, we could use enum.auto() or object() sentinels to produce members that represent distinct entities

recap



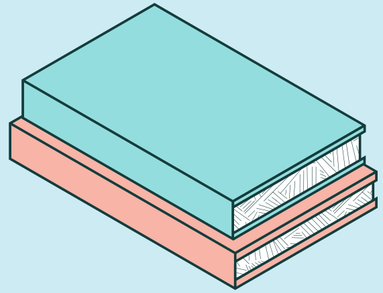
- auto() returns consecutive integers starting from 1 by default
- this behaviour could be customized by implementing the _generate_next_value_() method
- the method should be overridden before the members are defined
- typically this is done in a separate class from which the actual enum inherits from

recap



- enums in python are not just collections of symbolic names to constant values, but rather, complete types in their own right
- they could be extended with the right attributes or behaviour just like other classes
- we could also further subclass enums that have no members defined
- inheritance with enums is most useful in defining and sharing behaviour across the inheritance chain; we cannot subclass enums that already define members

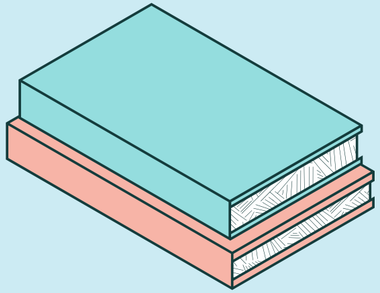
recap



- the enum.Flag class is a great choice for building enums that are meant to be combined to represent some sort of state

- Flag with auto sets the values to consecutive powers of 2 offering a simple yet powerful solution the problem of ambiguity and state value collision

recap



- the values that enum.Flag auto-assigns (i.e. consecutive powers of 2) unlock a powerful set of bitwise operations that are ideal for managing state that depends on a combination of several boolean states

- the most common bitwise operators and associated mnemonics are:

  | (OR), think of it as "union, or combination"

  & (AND) "intersection"

  ~ (NOT) "everything but"

  ^ (XOR) "in one or the other but not both"

Skill Challenge #12

#enumerations

Requirements

> Define a new type called Permission that stores user permissions: read, write, and/or execute

> This type should be an enumeration, with the ability to support bitwise operations

> Separately define a new type called User, which takes a name and user_role at instantiation

> Internally, the User class sets a permissions attribute depending on the specified user_group:
- admin: read, write, and execute
- user: read,
- manager: read, write
- support: execute

> The User class also implements (or ideally inherits) read(file), write(file, content), and execute(file) methods which are permission-checked, e.g. a User instance belonging to the support user_role will not be able to write, but only execute

> For ease of operation, assume that the read/write/execute functionality pertains to a python script

> Instances of User should have an informative string representation

> As an extra challenge, try to allow some polymorphism in the user_role so that it's possible to instantiate by both string roles as well as integers, e.g. User("A", user_role=2) would imply WRITE-only permissions, whereas User("B", user_role=6) would imply WRITE and EXEC, because 2**1 + 2**2 = 2 + 4 = 6