we could customize our objects' representation by implementing __repr__ in the objects' class definition

__repr__ vs __str__

tip

if you have to pick between the two, choose to define dunder repr because python falls to dunder repr if str is not defined

1. both dunder str and repr are used to define custom string representations for a given object
2. both dunder str and repr are used to define custom string representations for a given object
3. dunder repr ideally should return valid python code which, if evaluated, rebuilds the instance

recap

1. string representation formatting for our objects could be customized by defining __format__
2. f-strings, the format() built-in, and str.format() all call the same dunder
3. defining __format__ is completely optional; python will delegate to __str__ and, if that's not defined either, ultimately fallback to __repr__

recap

1. by default, instances of the same class in python are not equal
2. we customize object equality by defining __eq__

recap

1. a hash is just a fixed-length integer that identifies an object
2. an object is hashable if it
   - has a hash value that never changes over its life,
   - is comparable to other objects, and
   - shares the same hash value with objects it compares to as equal
3. immutable data types in python (e.g. int, float, str, tuple) are hashable
4. hashes are extremely useful in facilitating fast lookups in dictionaries and membership checks in sets; as a result, dictionary keys and set members need to be hashable

recap

1. by default, user-defined classes are hashable
2. when we define dunder eq, python (v3+) makes them unhashable mostly to protect us from unpleasant side effects
3. we make a class hashable again by defining dunder hash, which should should always return an integer
4. as it is closely related to equality, it's a good idea for dunder hash to consider the same attributes that dunder eq uses in determining equality

Skill Challenge #3

#dunders

Requirements

> Define a new type called Contact, which should store a person's name, last name, phone, email, as well as an instance attribute called *display_mode*, which defaults to "masked"

> Users should be able to create instances of Contact using name and last name only

> Two instances should be considered equal if any of the following conditions are met:

- phone or email are specified and the same, or
- name and last name are the same

> The instance representation should return obfuscated name and last name attributes when display_mode is set to "masked" and the regular full representation including all attributes otherwise

> The str() representation should always return the first letter of the last name followed by the first letter of the first name

> A user should be able to format a masked instance's string representation so as to reveal all the attributes

recap

1. in order to support the "<" and ">" operators we need to define one of __lt__ or __gt__
2. support for "≤" and "≥" needs to be defined separately through __le__ and __ge__

the total_ordering higher-order function from the functools module adds support for all comparison operators as long as we define

1) dunder eq, and

2) one of { gt, lt, ge, le }

recap

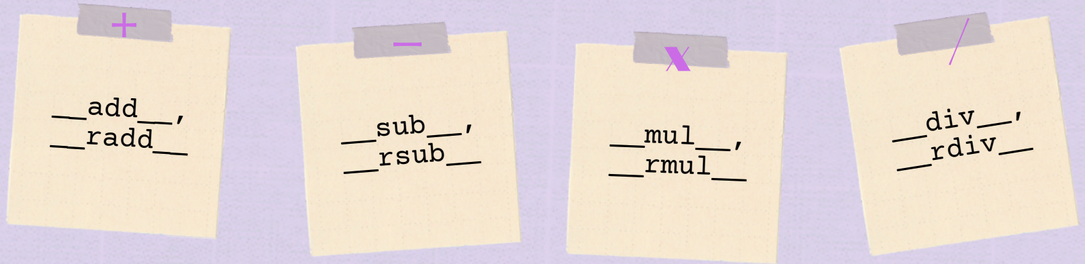
1. objects, including instances of user-defined classes, by default, are truthy
2. to customize this behavior, we define special logic within dunder bool

recap

1. classes could act as container abstractions for other classes
2. we built a container class for our Book instances that is capacity- and type-aware

recap

1. we could get our custom classes to support the plus operator by implementing __add__
2. to guarantee commutativity, __radd__ needs to be implemented too
3. other operators could be similarly overloaded to support user-defined classes



recap

1. we could add support for square bracket key lookup to our classes by implementing __getitem__
2. we are free to customize the functionality supported by adding further logic beyond the customary one based on integers and slice objects
3. implementing __getitem__ also makes our class iterable

recap

1. we could certainly define our own dunders, but we shouldn't
2. by convention, it's an antipattern, and
3. there is no guarantee that they will not clash with future python dunders

Skill Challenge #4

#dunders

Requirements

> Define a new type called Vector that stores 3 instance attributes: *x*, *y*, *z*

> Users should be able to create new instances as Vector(x=1, y=2, z=3), where the coordinates are positional args with no defaults

> Instances of this new Vector type should have a representation that would help the user reconstruct the instance

> The magnitude of the vector should be accessible through a method, ideally a built-in

- *hint*: the magnitude is calculated as sqrt of sum of squared coordinates
- *hint2*: as far as built-ins are concerned, __len__ will not work; try to target abs()?

> Users should be able to add two vectors to get a third, e.g. Vector(1, 2, 3) + Vector(4, 5, 6) -> Vector(5, 7, 9)

> Users should be able to numerically scale a vector, e.g. Vector(1, 2, 3) * 2 = Vector(2, 4, 6)

> The scalar multiplication operation should work the same regardless of the order of operands, e.g. Vector(1, 2, 3) * 2 = 2 * Vector(1, 2, 3)

Requirements Continued

> All comparison operators should be supported between two instances of Vector

> Vector should be hashable

> A Vector instance should evaluate to False if and only if its magnitude is zero

> Lastly, the Vector class should let the user select coordinates using square brackets too, e.g. if v1 = Vector(1, 2, 3) then both v1['y'] and v1['Y'] should return 2