Inheritance & Composition

SURP 2022 Python Bootcamp

Ohio State Astronomy

Slides by: James W. Johnson

What Are They?

Inheritance

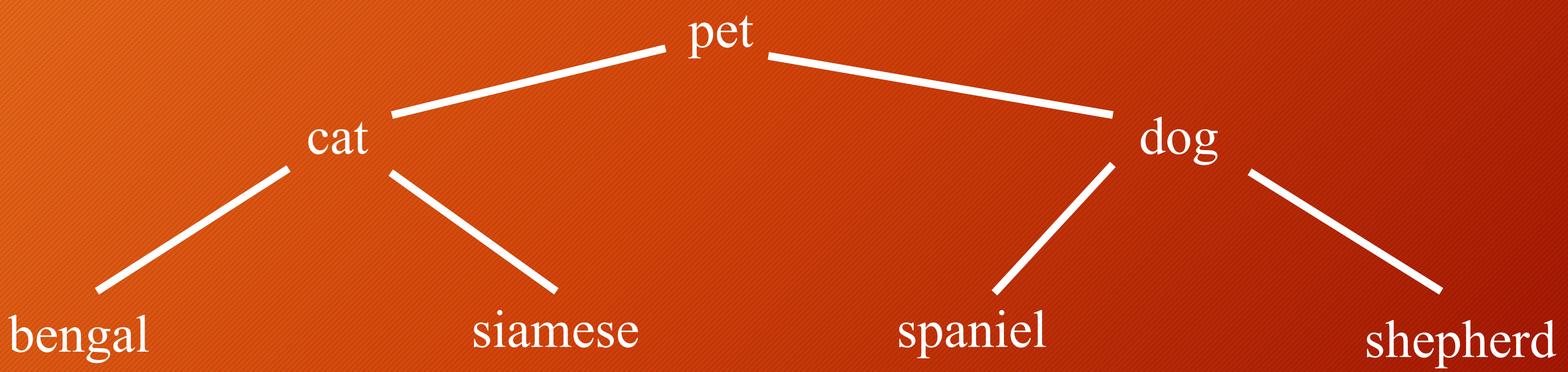
- An object is based on another object, is a special type of that object, or is some kind of extension of it
- *Subclasses* “inherit” functionality from their *parent class*, which may have its own parent class

Composition

- An object *contains* other objects, is *made of* them, and may not have meaning without them

Inheritance

A classic example: Pets (the implementation of this is left for an exercise)



Features shared by all pets should be implemented in the *pet* base class. Those shared by all cats but not dogs in the *cat* class. These features will be automatically included in all *subclasses* (a.k.a. *derived classes*).

Inheritance: Students and Professors

First: A base class

- You need something to inherit from

Both can inherit from a base class *person* storing the data any person would have (e.g. their name).

So far this is all material from last session.

```python
class person:

    def __init__(self, name):
        self.name = name

    @property
    def name(self):
        r"""
        Type : str

        The person's name.
        """
        return self._name

    @name.setter
    def name(self, value):
        if isinstance(value, str):
            # Capitalize first letter of their name
            self._name = value.capitalize()
        else:
            raise TypeError("Attribute 'name' must be a string. Got: %s" % (
                type(value)))
```

Inheritance: Students and Professors

Next: A derived class

The derived class needs to call its parent class's *__init__* function, and inheritance is accomplished.

```
class student(person):

    def __init__(self, name):
        super().__init__(name)
```

Here student objects have a property *name* which is *inherited* from the person object.

```
In [1]: from people import student

In [2]: mike = student("mike")

In [3]: peggy = student("peggy")

In [4]: mike.name
Out[4]: 'Mike'

In [5]: peggy.name
Out[5]: 'Peggy'

In [6]: from people import person

In [7]: isinstance(mike, person)
Out[7]: True

In [8]: isinstance(mike, student)
Out[8]: True
```

Inheritance: Students and Professors

Next: A derived class

The derived class can have properties, functions, etc. that is unique to that derived class.

```
class student(person):

    def __init__(self, name):
        super().__init__(name)

    def work(self):
        r"""
        Prints a message saying the student is working on their assignment.
        """
        print("%s is working on the assignment." % (self.name))
```

```
In [1]: from people import person, student

In [2]: mike = student("mike")

In [3]: peggy = person("peggy")

In [4]: mike.work()
Mike is working on the assignment.

In [5]: peggy.work()
---------------------------------------------------------------------------
AttributeError                            Traceback (mos
<ipython-input-5-8bcdb0ceea21> in <module>
----> 1 peggy.work()

AttributeError: 'person' object has no attribute 'work'
```

Inheritance: Students and Professors

Next: Another derived class

A different derived class can have a function with the same name executing a different task.

```python
class professor(person):

    def __init__(self, name):
        super().__init__(name)

    def work(self):
        r"""
        Prints a message saying the professor is grading the assignments.
        """
        print("%s is grading the assignments." % (self.name))
```

```
In [1]: from people import student, professor

In [2]: mike = student("mike")

In [3]: peggy = professor("peggy")

In [4]: mike.work()
Mike is working on the assignment.

In [5]: peggy.work()
Peggy is grading the assignments.
```

Inheritance: Students and Professors

The derived classes can also *override* inherited functions or properties.

```
class person:

    def __init__(self, name):
        self.name = name

    def work(self):
        r"""
        Prints a message saying the person is on the clock earning a paycheck.
        """
        print("%s is on the clock earning a paycheck." % (self.name))
```

This can be used to create objects which share data and do *different* things when you call the *same* function.

```
In [1]: from people import person, student, professor

In [2]: mike = student("mike")

In [3]: peggy = professor("peggy")

In [4]: kelly = person("kelly")

In [5]: mike.work()
Mike is working on the assignment.

In [6]: peggy.work()
Peggy is grading the assignments.

In [7]: kelly.work()
Kelly is on the clock earning a paycheck.
```

Inheritance: Subclassing Built-Ins

Built-in data types can be subclassed too!

The __*init*__ functions of built-in types usually accept **args* and ***kwargs* as parameters

Immutable types will require overriding __*new*__ as opposed to __*init*__

- This includes *int*, *float*, *bool*, *string*, *tuple*, and *range*
- In practice you'll only override __*new*__ in rare, specific instances
- We'll see an example of this in a few slides

Sometimes inheriting built-in features can be very powerful

Inheritance: Subclassing Built-Ins

Example: subclass *list* to make a simple *array*

Arrays differ from lists in that all elements must be of the same type.

```python
class simple_array(list):

    def __init__(self, *args, **kwargs):
        if "dtype" in kwargs.keys():
            self._dtype = kwargs["dtype"]
            del kwargs["dtype"]
        else:
            self._dtype = object

        if all([isinstance(i, self._dtype) for i in args[0]]):
            super().__init__(*args, **kwargs)
        else:
            raise TypeError("All elements must be of the specified data type.")

    def __setitem__(self, key, value):
        if isinstance(value, self.dtype):
            super().__setitem__(key, value)
        else:
            raise TypeError("Must be of type %s. Got: %s" % (self.dtype,
                type(value)))

    @property
    def dtype(self):
        r"""
        Type : type

        The data type of the elements.
        """
        return self._dtype
```

Inheritance: Subclassing Built-Ins

In the example, a *TypeError* is raised by __*init*__ if not all elements are of the specified *dtype*.

Then override the inherited __*setitem*__ to only accept the specified *dtype*.

Note the reappearance of *super* – it can be used *anywhere* to refer to an inherited class or function.

```
class simple_array(list):

    def __init__(self, *args, **kwargs):
        if "dtype" in kwargs.keys():
            self._dtype = kwargs["dtype"]
            del kwargs["dtype"]
        else:
            self._dtype = object

        if all([isinstance(i, self._dtype) for i in args[0]]):
            super().__init__(*args, **kwargs)
        else:
            raise TypeError("All elements must be of the specified data type.")

    def __setitem__(self, key, value):
        if isinstance(value, self.dtype):
            super().__setitem__(key, value)
        else:
            raise TypeError("Must be of type %s. Got: %s" % (self.dtype,
                type(value)))

    @property
    def dtype(self):
        r"""
        Type : type

        The data type of the elements.
        """
        return self._dtype
```

Inheritance: Subclassing Built-Ins

All of the features and behavior of the list are inherited, with the modification that this only allows integers

```
In [2]: example = simple_array([0, 1, 2], dtype = int)

In [3]: example
Out[3]: [0, 1, 2]

In [4]: example[0]
Out[4]: 0

In [5]: example[0] = 1

In [6]: example
Out[6]: [1, 1, 2]

In [7]: example[0] = 1.1
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent
<ipython-input-7-21f8ac3b2c15> in <module>
----> 1 example[0] = 1.1

~/Work/Teaching/OSUAstroSURP2020/examples/classes/simple_array.py
y, value)
     26                 else:
     27                         raise TypeError("Must be of type
pe,
---> 28                                 type(value)))
     29
     30         @property

TypeError: Must be of type <class 'int'>. Got: <class 'float'>
```

This Includes Exceptions and Warnings

You can create your own *Exception* and *Warning* classes by subclassing these built-in types

Unless you want to do something special, this only requires two lines

```
In [1]: import warnings

In [2]: class CustomError(Exception):
   ...:     pass
   ...:

In [3]: class CustomWarning(Warning):
   ...:     pass
   ...:

In [4]: raise CustomError("My custom error class.")
---------------------------------------------------------------------------
CustomError                               Traceback (most recent call last)
<ipython-input-4-5dab39228a42> in <module>
----> 1 raise CustomError("My custom error class.")

CustomError: My custom error class.

In [5]: warnings.warn("My custom warning class.", CustomWarning)
/Users/astrobeard/anaconda3/bin/ipython:1: CustomWarning: My custom warning class.
  #!/Users/astrobeard/anaconda3/bin/python
```

Inheritance: Subclassing Built-Ins

Sub-classing immutable types requires over-riding __*new*__ rather than __*init*__ (example: positive *int*)

__*new*__ vs. __*init*__:

- __*init*__ must return *None* ; __*new*__ must return the object
- *cls* versus *self* as first parameter
- __*new*__ handles creation of the class ; __*init*__ handles initialization
- __*new*__ can be used to return instances of entirely different classes if need be

```
class positive(int):
    """
    A positive definite integer.
    """

    def __new__(cls, value, *args, **kwargs):
        if value <= 0: raise ValueError("Must be non-negative and non-zero.")
        return super().__new__(cls, value, *args, **kwargs)

    def __add__(self, other):
        return self.__class__(super().__add__(other))

    def __sub__(self, other):
        return self.__class__(super().__sub__(other))

    def __mul__(self, other):
        return self.__class__(super().__mul__(other))

    def __div__(self, other):
        return self.__class__(super().__div__(other))

    def __str__(self):
        return "%d" % (int(self))

    def __repr__(self):
        return "positive(%d)" % (int(self))
```

Inheritance: Subclassing Built-Ins

Sub-classing immutable types requires over-riding *__new__* rather than *__init__* (example: positive *int*)

__new__ vs. *__init__*:

- *__init__* must return *None* ; *__new__* must return the object
- *cls* versus *self* as first parameter
- *__new__* handles creation of the class ; *__init__* handles initialization
- *__new__* can be used to return instances of entirely different classes if need be

```
In [1]: from positive import positive

In [2]: example = positive(3)

In [3]: example
Out[3]: positive(3)

In [4]: example + 4
Out[4]: positive(7)

In [5]: example - 5
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-b6c0a2daceee> in <module>
----> 1 example - 5

~/Work/Teaching/PythonBootcamp/examples/classes/positive.py in __sub__(self, other)
     14
     15     def __sub__(self, other):
---> 16         return self.__class__(super().__sub__(other))
     17
     18     def __mul__(self, other):

~/Work/Teaching/PythonBootcamp/examples/classes/positive.py in __new__(cls, value, *args, **kwargs)
      7
      8     def __new__(cls, value, *args, **kwargs):
----> 9         if value < 0: raise ValueError("Must be non-negative and non-zero.")
     10         return super().__new__(cls, value, *args, **kwargs)
     11

ValueError: Must be non-negative and non-zero.
```

Recall: The Python Model

Everything is an object

At the end of the day, *everything* inherits from *object*

isinstance(x, object) will ***always*** return *True*

```
In [1]: isinstance(3, object)
Out[1]: True

In [2]: isinstance("some string", object)
Out[2]: True

In [3]: import numpy

In [4]: isinstance(numpy, object)
Out[4]: True

In [5]: isinstance([0, 1, 2], object)
Out[5]: True
```

Composition

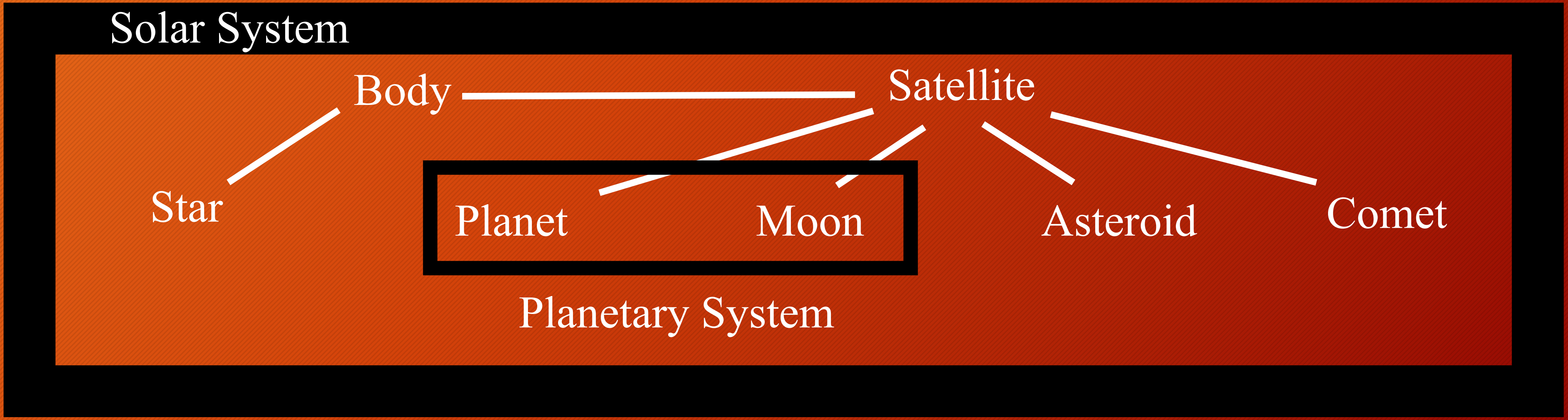
An astronomical example: A solar system

Components: star, planets, moons, asteroids, comets

Composition

An astronomical example: A solar system

Components: star, planets, moons, asteroids, comets



Composition: A Solar System

First pieces: the inheritance structure of the solar system bodies. All solar system bodies have a name and a mass, so we put those in the base class.

```python
class body:

    def __init__(self, name, mass):
        self.name = name
        self.mass = mass

    @property
    def name(self):
        r"""
        Type : str

        The name of the star
        """
        return self._name

    @name.setter
    def name(self, value):
        if isinstance(value, str):
            self._name = value
        else:
            raise TypeError("Attribute 'name' must be a string. Got: %s" % (
                type(value)))

    @property
    def mass(self):
        r"""
        Type : float

        The mass of the star in solar masses
        """
        return self._mass

    @mass.setter
    def mass(self, value):
        if isinstance(value, numbers.Number):
            self._mass = float(value)
        else:
            raise TypeError("Attribute 'mass' must be a real number. Got: %s" % (
                type(value)))
```

Composition: A Solar System

First pieces: the inheritance structure of the solar system bodies. All solar system bodies have a name and a mass, so we put those in the base class.

The star doesn't need any more than this, so we can let it inherit *everything* – even the __*init*__ function.

```
class star(body):
    pass
```

Composition: A Solar System

First pieces: the inheritance structure of the solar system bodies. All solar system bodies have a name and a mass, so we put those in the base class.

A satellite object also has a semi-major axis and an eccentricity.

```
class satellite(body):

    def __init__(self, name, mass, semimajor_axis, eccentricity = 0):
        super().__init__(name, mass)
        self.semimajor_axis = semimajor_axis
        self.eccentricity = eccentricity

    @property
    def semimajor_axis(self):
        r"""
        Type : float

        The semimajor axis of the satellite's orbit in AU
        """
        return self._semimajor_axis

    @semimajor_axis.setter
    def semimajor_axis(self, value):
        if isinstance(value, numbers.Number):
            if value > 0:
                self._semimajor_axis = value
            else:
                raise ValueError("""Attribute 'semimajor_axis' must be \
positive. Got: %f""" % (value))
        else:
            raise TypeError("""Attribute 'semimajor_axis' must be a real \
number. Got: %s""" % (type(value)))

    @property
    def eccentricity(self):
        r"""
        Type : float

        The eccentricity of the satellite's orbit
        """
        return self._eccentricity

    @eccentricity.setter
    def eccentricity(self, value):
        if isinstance(value, numbers.Number):
            if 0 <= value <= 1:
                self._eccentricity = value
            else:
                raise ValueError("""Attribute 'eccentricity' must be between
0 and 1. Got: %f""" % (value))
        else:
            raise TypeError("""Attribute 'eccentricity' must be a real \
number. Got: %s""" % (type(value)))
```

Composition: A Solar System

First pieces: the inheritance structure of the solar system bodies. All solar system bodies have a name and a mass, so we put those in the base class.

A satellite object also has a semi-major axis and an eccentricity.

Planets, moons, asteroids, and comets don't need any data beyond that.

```python
class planet(satellite):
    pass

class moon(satellite):
    pass

class asteroid(satellite):
    pass

class comet(satellite):
    pass
```

Composition: A Solar System

Next: A planetary system composed of a planet object and moons. This requires no new syntax – these can be properties of a new class.

Here the planet attribute is just the planet object, and the moons is a list of moon objects. Composition introduces no new syntax - it just refers to properties of a specific type.

```
class planetary_system:

    def __init__(self, planet, moons):
        self.planet = planet
        self.moons = moons

    @property
    def planet(self):
        r"""
        Type : planet

        The central planet
        """
        return self._planet

    @planet.setter
    def planet(self, value):
        if isinstance(value, planet):
            self._planet = value
        else:
            raise TypeError("""Attribute 'planet' must be of type planet. \
Got: %s""" % (type(value)))

    @property
    def moons(self):
        r"""
        Type : list

        A list of moon objects which orbit the planet
        """
        return self._moons

    @moons.setter
    def moons(self, value):
        if isinstance(value, list):
            if all([isinstance(i, moon) for i in value]):
                self._moons = value[:]
            else:
                raise TypeError("All moons must be of type moon.")
        else:
            raise TypeError("""Attribute 'moons' must be of type list. \
Got: %s""" % (type(value)))
```

Composition: A Solar System

Next: A solar system object composed of a star, planetary system objects, asteroids, and comets.

Note: the __*init*__ function is calling setter functions not pictured here.

```
class solar_system:

    def __init__(self, star, planets, asteroids, comets):
        self.star = star
        self.planets = planets
        self.asteroids = asteroids
        self.comets = comets

    @property
    def star(self):
        r"""
        Type : star

        The central star of the solar system.
        """
        return self._star

    @star.setter
    def star(self, value):
        if isinstance(value, star):
            self._star = value
        else:
            raise TypeError("Attribute 'star' must be of type star. Got: %s" % (
                type(value)))
```

Composition: A Solar System

Next: A solar system object composed of a star, planetary system objects, asteroids, and comets.

The *planets* attribute is a list of *planetary_system* objects. The *asteroids* and *comets* properties proceed similarly.

```python
@property
def planets(self):
    r"""
    Type : list

    A list of planetary_system objects which orbit the central star.
    """
    return self._planets

@planets.setter
def planets(self, value):
    if isinstance(value, list):
        if all([isinstance(i, planetary_system) for i in value]):
            self._planets = value
        else:
            raise TypeError("All planets must be of type planetary_system.")
    else:
        raise TypeError("""Attribute 'planets' must be of type list. \
Got: %s""" % (type(value)))
```

Composition: A Solar System

The *planetary_system* and the *solar_system* objects in action:

```
In [1]: import solar_system

In [2]: sun = solar_system.star("sun", 1)

In [3]: earth = solar_system.planet("earth", 3.e-6, 1)

In [4]: moon = solar_system.moon("luna", 3.e-8, .003)

In [5]: mars = solar_system.planet("mars", 3.e-7, 1.5)

In [6]: phobos = solar_system.moon("phobos", 5.3e-15, 6.3e-5)

In [7]: deimos = solar_system.moon("deimos", 7.4e-16, 1.6e-4)

In [8]: earth_system = solar_system.planetary_system(earth, [moon])

In [9]: mars_system = solar_system.planetary_system(mars, [phobos, deimos])
```

```
In [10]: our_solar_system = solar_system.solar_system(sun, [earth_system, mars_system], [], [])

In [11]: our_solar_system.star.name
Out[11]: 'sun'

In [12]: our_solar_system.planets[0].planet.name
Out[12]: 'earth'

In [13]: our_solar_system.planets[0].moons[0].name
Out[13]: 'luna'

In [14]: our_solar_system.planets[1].planet.name
Out[14]: 'mars'

In [15]: our_solar_system.planets[1].moons[0].name
Out[15]: 'phobos'

In [16]: our_solar_system.planets[1].moons[1].name
Out[16]: 'deimos'

In [17]: our_solar_system.asteroids
Out[17]: []

In [18]: our_solar_system.comets
Out[18]: []
```

Composition vs. Aggregation

Composition differs in detail from *aggregation* – composition implies ownership whereas aggregation implies usage – the two are often confused. Neither involve special syntax.

Consider a program with two objects: A and B

- Composition: A "owns" B, and B is destroyed when A is destroyed
- Aggregation: A "uses" B, and B is *not* destroyed when A is destroyed

Real world example: ammunition in a shooter video game

- If it's dropped when you die, and another player can pick it up → aggregation
- If it's not dropped when you die, instead disappearing from the game → composition