OOP - Inheritance

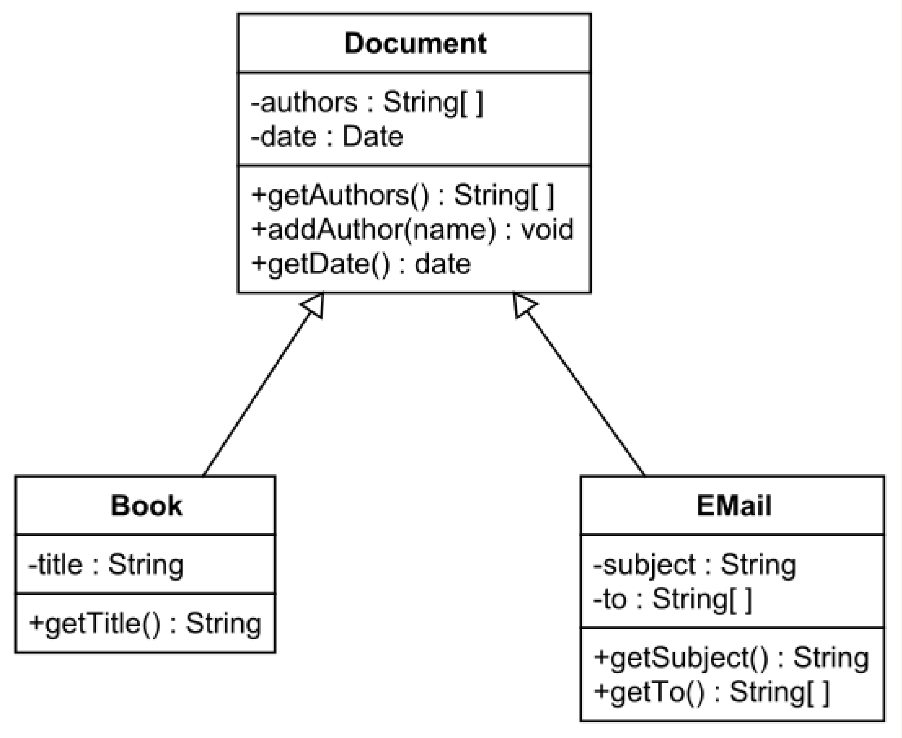
Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

Inheritance

Inheritance is the mechanism of building a class upon another class, maintaining a similar implementation.

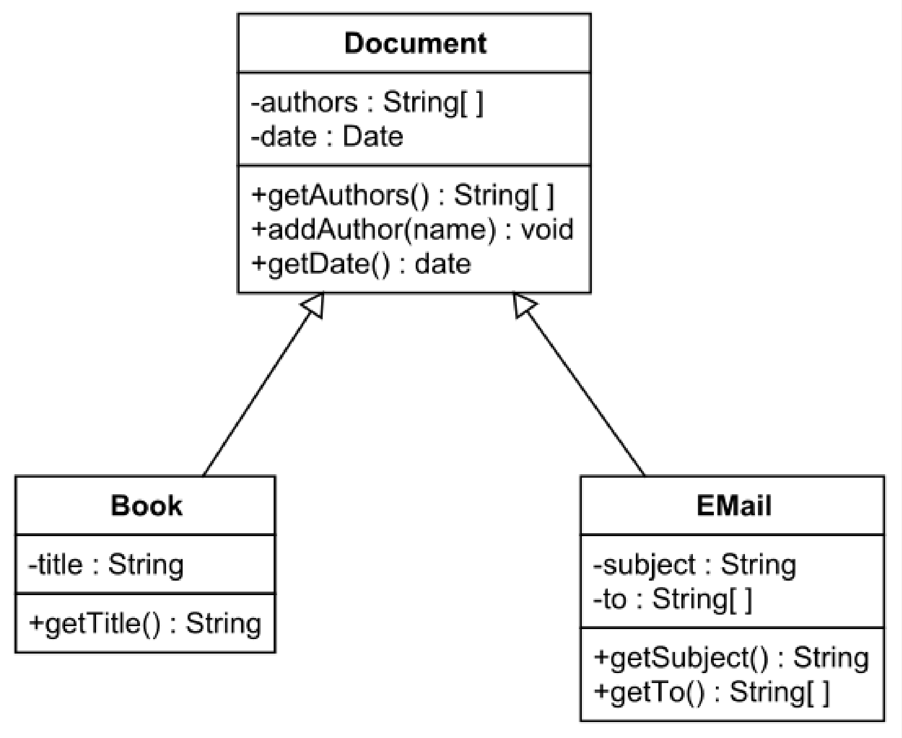
- The 'original' class is called Superclass, base class, or parent class
- The 'new' class is called Subclass, derived class, or child class



Inheritance

A class **subclasses** another class.
The `Book` class and the `Mail` class subclass the `Document` class.

The Book and Email classes **inherit the attributes and methods** from the Document class. It is possible to modify the methods and **add new fields and methods**.



Main Forms of Inheritance

- **Specialization**: subclasses override methods from the superclass, keeping their other features.
- **Specification**: an *abstract* superclass defines methods that will be implemented by its subclasses.
- **Extension**: subclasses add new methods without altering inherited attributes/methods.
- **Combination**: a subclass inherit from multiple classes.

Inheritance - (*Python implementation*)

All Python classes are subclasses of the special class named `object`.

So all Python classes inherit all methods that we use, but we are usually not aware of them, such as the `__new__` class method.

```
class MySubClass1(object):
  pass


class MySubClass2:
  pass
```

Add new behavior to existing class

To define new behaviors we can add new methods to the subclass.

```python
class MyClass:

  def __init__(self):
    # doSomething()
    ...

  def f1(self):
    ...


class MySubClass(MyClass):

  # it uses the __init__ and f1 method of the superclass

  def f2():
    ...
```

Change behavior to existing class

To change the behavior we can redefine (*override*) a method in the subclass.

```
class MyClass:

  def __init__(self):
    # doSomething()
    ...

  def f1(self):
    ...


class MySubClass(MyClass):

  # it uses the __init__  method of the superclass

  def f1(self):  # redefine the method
    ...
```

Change behavior to existing class

Sometimes we want the new method to do what the old method did, **plus other actions**.

```
class MyClass:

  def f1(self):
    # do_1()
    # do_2()


class MySubClass(MyClass):

  def f1(self):  # redefine the method
    # do_1()  # Problem: duplicate code
    # do_2()  # Problem: duplicate code
    # do_3()
```

Change behavior to existing class

Code maintenance is complicated. We have to update the code in two or more places. We need a way to **execute the original `f1()` method** on the `MyClass` class, and after the **new `f1()` method**.

```
class MyClass:

  def f1(self):
    # do_1()
    # do_2()


class MySubClass(MyClass):

  def f1(self):  # redefine the method
    super().f1()
    # do_3()
```

The `super()` function returns an instance of the parent class, allowing us to call the parent method directly.

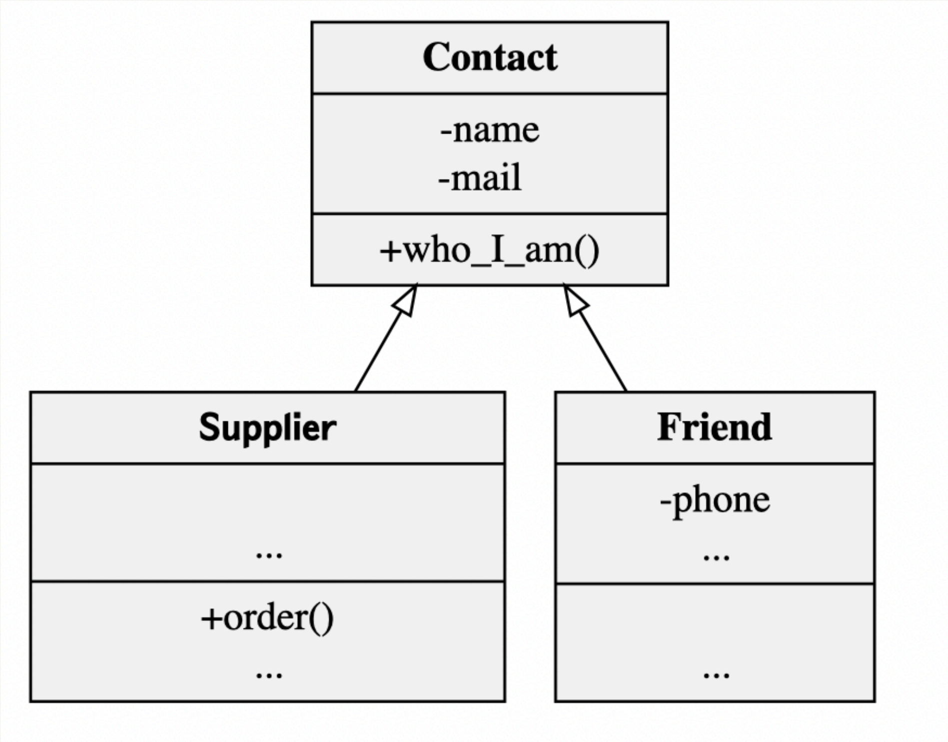
A `super()` call can be made inside any method.
All methods can be modified via overriding and calls to `super()`.

Example

- Define a `Contact` class. Instances have attributes **name** and a **email**.
- When the object is instantiated, the name and email are initialized, and the formal correctness of the email address is checked[(1)].
- Instances have a method `who_I_am()` that returns a string composed of name and email.

[(1)]: Trivial approach: email address must contain `@`, with other characters before and after `@`. Use the String `split()` method.

For a complete solution, you can use the **regular expression** module `re` - https://docs.python.org/3/library/re.html#



- `Supplier` (subclass of `Contact`) has a method `order()` to place purchase orders (*the method merely prints a string*).
- `Friend` (subclass of `Contact`) stores the phone number during its creation.

```python
c = Contact("pippo", "pippo@gmail.com")
s = Supplier("pluto", "pluto@gmail.com")

print(c.name , c.email)
print(s.name , s.email)

# c.order("mouse")
# AttributeError: "Contact" object has no attribute "order"
s.order("mouse")

f = Friend("goofie", "goofie@gmail.com", "123123")
```

Solution

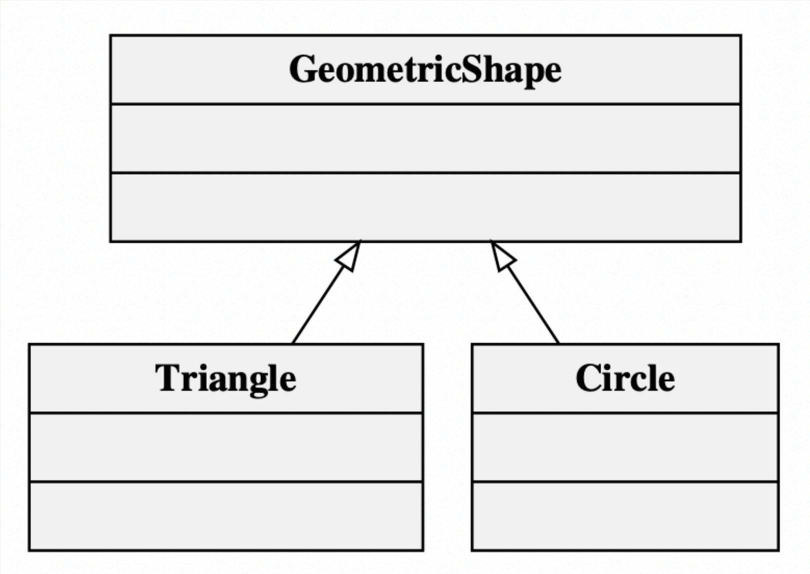
```
class Contact():

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

  @property
  def name(self):
    return self._name

  @name.setter
  def name(self, name):
    self._name = name

  @property
  def email(self):
    return self._email
```

Abstract classes

Sometimes it makes **no sense to instantiate objects from the superclass**. For example, while it makes sense to instantiate objects from the `Triangle` or `Circle` classes, it makes no sense to instantiate objects from the `GeometricShape` class.

However, it is useful to have a `GeometricShape` superclass:

- to standardize the interfaces of the subclasses
- to factor the code (DRY)



Abstract classes

An abstract class is a class from which **objects cannot be instantiated**.

An **abstract method** is a method that is *decorated* with the `@abstractmethod` keyword.

Abstract classes are subclasses of the `ABC` class and contain one or more abstract methods. A class that is derived from an abstract class cannot be instantiated unless **all of its abstract methods are overridden**.

An abstract class indicates the common interface of all subclasses and specifies which methods must be overridden.

Example

```python
from abc import ABC, abstractmethod


class Character(ABC):

def __init__(self, name):
  self.name = name
  self.score = 0

def fight(self):
  print("THIS IS SPARTA!")

@abstractmethod
def jump(self):  # you are forced to redefine this method
  pass
```

An abstract method can have an implementation in the abstract class, but designers of subclasses will be forced to override the implementation.
It is possible to provide some basic functionality in the abstract method, which can be enriched by the subclass implementation.

Example (`jump` is the common part of the message):

- A mouse must print the message
  JUMP!
  A mouse can jump
- A kangaroo must print the message
  JUMP!
  A kangaroo can jump very high

Solution

```python
from abc import ABC, abstractmethod


class Character(ABC):

  def __init__(self, name):
    self.name = name
    self.score = 0

  def fight(self):
    print("THIS IS SPARTA!")

  @abstractmethod
  def jump(self):  # you are forced to redefine this method
    print("JUMP!")


class Mouse(Character):
```

Using abstract classes we can

- specify that the method `m()` of the class `C` must be (re)defined in each subclass. This can be done by making `C` an abstract class with the `m()` method declared abstract.
- make a class that does not implement any methods. Such classes are called **interfaces**.

What we mentioned about abstract methods can also be applied to the `__init__()` method.

Consider two classes `A` and `B` whose instances have attributes
`a = 0`, `z = 0` for objects of class A
`b = 0`, `z = 0` for objects of class B

The initialization takes place during the creation of the object.

(trivial solution, with duplicate code)

```python
class A:

  def __init__(self):
    self.a = 0
    self.z = 0


class B:
  def __init__(self):
    self.b = 0
    self.z = 0
```

Solution: force subclasses to instantiate their **init**() method.

```python
from abc import ABC, abstractmethod


class Z(ABC):
  @abstractmethod
  def __init__(self):
    self.z = 0


class A(Z):
  def __init__(self):
    super().__init__()
    self.a = 0


class B(Z):
  def __init__(self):
    super().__init__()
    self.b = 0
```

Liskov substitution principle

The principle states that **objects of a subclass can replace objects of the superclass without breaking the application**.

Functions or methods that use objects of a base classes must be able to use objects of derived classes without knowing it.

The objects of the subclass must behave like the objects of the superclass.
An overridden method in a subclass needs to accept the same input parameter as the method in the superclass.

Similar rules apply to the return value of the method.
The return value of a method of the subclass needs to comply with the same rules as the return value of the method of the superclass.

https://en.wikipedia.org/wiki/Liskov_substitution_principle

Liskov substitution principle - example

Class B objects **cannot** replace class A objects.

```
class A:

  def f(self, x):
    print(x)

  def g(self, x):
    print (x)


class B(A):

  def f(self):
    pass

  def g(self, x, y):
    print (x, y)

  def h(self, x, y):
    ...


obj = A()   # what happens if obj = B()?
obj.f(10)
obj.g(10)
```

Liskov substitution principle - example

Class B objects **cannot** replace class A objects.

```python
class A:

  def f(self, x):
    print(x)

  def g(self, x):
    print (x)


class B(A):

  def f(self):
    pass

  def g(self, x, y):
    print (x, y)

  def h(self, x, y):
    ...


obj = A()  # what happens if obj = B()?
obj.f(10)
obj.g(10)
```

Now class B objects **can** replace class A objects.

```python
class A:

  def f(self, x):
    print(x)

  def g(self, x):
    print (x)


class B(A):

  def f(self, x=0):
    pass

  def g(self, x, y=0):
    # same interface!
    print (x, y)

  def h(self, x, y):
    ...


obj = A()  # what happens if obj = B()?
obj.f(10)
obj.g(10)
```

Another example of violation

Define a hierarchy in which the `Sparrow` and `Penguin` subclasses derive from the `Bird` superclass. The `Bird` superclass exposes the (abstract) `fly` method.

```python
# Classes...

# Function that takes a Bird and expects it to fly
def make_bird_fly(bird):
  bird.fly()


b1 = Sparrow()
b2 = Penguin()
list_of_birds = [b1, b2]
for el in list_of_birds:
  make_bird_fly(el)  # Raises ValueError: "I can't fly!"
```

Solution

```python
from abc import ABC, abstractmethod


class Bird(ABC):

  @abstractmethod
  def fly(self):
    pass


class Sparrow(Bird):
  def fly(self):
    print("I can fly!")


class Penguin(Bird):
  def fly(self):
    # Penguins cannot fly, so this violates LSP
```

If the superclass `Bird` exposes the `fly` method, it means that all the objects of the subclasses must respond to the `fly` method, even if in different ways (flapping flight, gliding flight...) This is the agreement with the Client.

The **Liskov substitution principle** requires fulfilling this agreement for all the subclasses. We **can** delete the method (`raise ValueError`) or merely print a message (`I can't fly!`), but in this way we are breaking the initial agreement.

A possible solution:

- the `Bird` superclass does not expose the `fly` method. `Penguin` is directly descended from `Bird`, so we don't expect it to have the fly method. The `Bird` class is further specialized into the `FlyingBird` class, from which `Sparrow` descends.

```python
from abc import ABC, abstractmethod


class Bird(ABC):
  pass


class FlyingBird(ABC):
  def fly(self):
    print("I can fly!")


class Sparrow(FlyingBird):
  pass

class Penguin(Bird):
  pass
```

Square and Rectangle, a more subtle violation

- Write a class **Rectangle** with sides (*attributes*) `a`, `b`. Write property and setters.
- Write a subclass **Square** (sides must be identical)

NB: you must override both `@property` and `@<property_name>.setter` in the subclass.

The proposed interface allows setting the two sides of the geometric figure separately. This leads to problems similar to what we observed with the TimeSlot class.

```python
print("rectangle")
r1 = Rectangle(1, 2)
print(r1.a, r1.b)  # 1, 2
r1.a = 10  # 10, 2
print(r1.a, r1.b)
```

```python
print("square")
s1 = Square(2)
print(s1.a, s1.b)  # 2, 2
s1.a = 10
print(s1.a, s1.b)  # 10, 10
s1.b = 20
print(s1.a, s1.b)  # 20, 20
```

Solution

```
class Rectangle:

  def __init__(self, a, b):
    self._a = a
    self._b = b

  @property
    def a(self):
      return self._a

  @a.setter
  def a(self, value):
    self._a = value

  @property
  def b(self):
    return self._b

  @b.setter
    def b(self, value):
```

Solution

```
class Square(Rectangle):

  # Overriding the setters to maintain the square property

  def __init__(self, a, b=None):
    super().__init__(a, a)

  @property
  def a(self):
    return self._a

  @a.setter
  def a(self, value):
    self._a , self._b = value, value

  @property
  def b(self):
    return self._b

  @b.setter
```

Now, the `Square` object behaves just like a square. It always has equal sides.

The model *seems* to be self-consistent, but it is not consistent with all its uses!

A model, viewed in isolation, can not be validated. The validity of a model can only be expressed in terms of its clients.

Add a **method** to compute the area. Define a **function** `check_area(shape, a, b)` that checks area, designed for a `Rectangle` object. What happens?

Solution

```python
# in the Superclass
def area(self):
  return self.a * self.b


# function:
def check_area(shape, a, b):
  shape.a = a
  shape.b = b
  return shape.area() == a * b


# client:
r = Rectangle(1,1)
s = Square(1)
print(check_area(r, 5, 6))
print(check_area(s, 5, 6))
```

A square is a rectangle, but a `Square` object is not a `Rectangle` object.

The behavior of a `Square` object is inconsistent with the behavior of a `Rectangle` object.

Possible solutions:

- make the objects immutable
- make it possible to set the lengths of the sides together, not separately, through a `set_sides` method that checks the correctness of the input values.

Examples of violation

1. The overridden method has a different number of *mandatory* arguments than those of the superclass method.
2. the overridden method prohibits previously permitted behavior
3. the interface is apparently the same, but the internal coherence of the object is destroyed

Composition

Composition is the act of collecting several objects together to create a new one.
Typically we have two ways to extend the functionality of a class: **inheritance** and **composition**.

Composition over inheritance is the principle that suggests the use of composition, where classes contain instances of other classes to implement desired functionality, instead of relying on inheritance from a base class.
This approach enhances code reuse by assembling existing components.

- In some scenarios, *inheritance* is the optimal solution: **the Orange class extends the Fruit class**.
- In other cases, *composition* provides a more accurate model of the system: **a car is composed of an engine, four wheels**, and so on[2].

Composition establishes a *has-a* relationship, in contrast to the *is-a* relationship in subclassing.

---

[2] From a syntactic point of view it is possible to extend (using inheritance) the class Engine by adding attributes representing `wheels`, `seats`, `steering wheel`, and so on. While we can think of a car as *an engine with wheels and seats*, this doesn't model reality well. It is more consistent to think of the car as the *union of the various parts that compose it*.

Exercise (1)

Write a class `Car` using **inheritance** (a `Car` **is** an `Engine` that has also `seats` and `wheels`). Attributes **displacement** and **n_seats** are Integers. **wheels_pressure** is a list of numbers.

```python
# Client

car = Car(2000, 4, 7)  # displacement, n_wheels, pressure, n_seats

print("displacement:", car.displacement)
print("wheel pressure:", car.wheels_pressure)  # [7, 7, 7, 7]

car.set_wheel_pressure(0, 1)
car.set_wheel_pressure(1, 2)
car.set_wheel_pressure(2, 3)
car.set_wheel_pressure(3, 4)
car.set_wheel_pressure(4, 5)  # NO ACTION - wheels are 0,1,2,3

print("wheel pressure:", car.wheels_pressure)  # [1, 2, 3, 4]
print("n seats:", car.n_seats)
```

Solution

```
class Engine:

  def __init__(self, displacement=1000):
    self.displacement = displacement


class EngineWithWheels(Engine):

  def __init__(self, displacement=1000, n_wheels=4, pressure=100):
    super().__init__(displacement)
    self.n_wheels = n_wheels
    self.wheels_pressure =  [pressure] * n_wheels

  def set_wheel_pressure(self, i_w, p):
    if i_w in range(0, self.n_wheels):
      self.wheels_pressure[i_w] = p

class Car(EngineWithWheels):

  def __init__(self, displacement=1000,n_wheels =4, pressure=100, n_seats=5):
    super().__init__(displacement, n_wheels, pressure)
```

Exercise (2)

Write a class `Car` using **composition** (a `Car` is a **composition of** `engine`, `seats` and `wheels`). Now, **engine**, **wheels** and **seats** are objects created outside of the CAR object.

Interface modification: The `Car` instantiation process is altered, with the `engine` now represented as a distinct object, and `wheels` and `seats` presented as lists of objects. Use the previous interface to display values.

```python
# Client
engine = ...  # create an engine
wheels = ...  # create a list of wheels
seats = ...  # create a list of seats
car = Car(engine, wheels, seats)
```

```python
print("displacement:", car.displacement)
print("wheel pressure:", car.wheels_pressure)  # [7, 7, 7, 7]

car.set_wheel_pressure(i, p)  # i in [0, 1, 2, 3]

print("wheel pressure:", car.wheels_pressure)  # [1, 2, 3, 4]
print("n seats:", car.n_seats)  # 5
```

Solution

```python
class Engine:

  def __init__(self, displacement=1000):
    self.displacement = displacement


class Wheel:

  def __init__(self, pressure=10):
    self.pressure = pressure


class Seat:

  def __init__(self, color=1):  # 1, 2, 3
    self.color = color


class Car:
```

Exercise (3)

Instantiate two cars and change the pressure of one wheel of `car 1`.

```
car1 = Car(engine, wheels, seats)
car2 = Car(engine, wheels, seats)

car1.set_wheel_pressure(0,5)

print("WHEEL - PRESSURE after that we change
      the pressure of WHEEL 0 - CAR 1")
print("car 1:", car1.wheels_pressure)
print("car 2:", car2.wheels_pressure)
```

What happens to the **other** car? (If we don't use **deepcopy**(3), cars share the same 'wheel' object!)

(3) https://docs.python.org/3.7/library/copy.html

Solution

```
class Car:

  def __init__(self, engine, wheels, seats):
    self.engine = engine
    self.wheels = copy.deepcopy(wheels)
```

A valid alternative is to provide the parameters needed to construct the other objects as input to `Car`.

The `__init__` method of `Car` takes care of instantiating objects, rather than receiving them 'ready-made' from the client.

Lazy initialization

Is a design pattern in which the creation of an object or the computation of a value is delayed until it is actually needed.
Instead of initializing the object or computing the value when the program starts or when an instance is created, lazy initialization waits until the first time the object or value is requested.

This can be useful in scenarios where the cost of initialization is high, and you want to defer it until it's necessary to improve performance or resource efficiency.

The key idea behind lazy initialization is to postpone the initialization process until the point where the initialized object or value is required for some specific operation.

Example of Lazy initialization

In the `Car` example the value `n_wheels` is computed from the list `wheels`. The Lazy initialization can be applied.

```python
class Car:

  def __init__(self, engine, wheels, seats):
    self._n_wheels = None
    # ...

  @property
  def n_wheels(self):

    # LAZY INITIALIZATION
    if self._n_wheels is None:
      self._n_wheels = len(self.wheels)
    return self._n_wheels
```

By defining a property for `n_wheels` without defining a setter, we make the `n_wheels` attribute read-only. It cannot be set from outside using `car._n_wheels = ...`.
This is appropriate since it is a parameter that can only be derived from the length of the `wheels` list. It makes no sense to set a different value.

Exercise

Design a class whose instances can store a list of items. Items can be added individually using the `add_item()` method, or in batches through the `add_collection()` method.

The object should **store the first element, skip the second one, store the third**, and so on. The object should only respond to the methods `add_item()` and `add_collection()`.

Implement the classes using:

1. only inheritance (class `AlternateList_inh(list)`)
2. only composition (class `AlternateList_comp`)

```python
obj = AlternateList()
obj.add_item(...)
obj.add_collection(...)

"""
Example:
  input items: 10, 20, 30, 40
  stored items: 10, 30
  What happens with obj.append() or obj.extend()?
"""
```

Test the classes with this client code:

```
n = 21
obj_comp = AlternateList_comp()
obj_inh = AlternateList_inh()
for i in range(1, n):
  obj_comp.add_item(i)
  obj_inh.add_item(i)
print(obj_comp)
print(obj_inh)

obj_comp = AlternateList_comp()
obj_inh = AlternateList_inh()
obj_comp.add_collection(list(range(1, n)))
obj_inh.add_collection(list(range(1, n)))

print(obj_comp)
print(obj_inh)
```

Solution

```python
class AlternateListComp:

  def __init__(self):
    self.values = []
    self.accept = True

  def __repr__(self):
    return str(self.values)

  def add_item(self, v):
    if self.accept:
      self.values.append(v)
    self.accept = not self.accept

  def add_collection(self, v_collection):
    for v in v_collection:
      self.add_item(v)
```

Solution

```
class AlternateListInh(list):

  def __init__(self):
    # super().__init__ ...
    # if you need to run even the init of the superclass
    self.accept = True

  def add_item(self, v):
    if self.accept:
      self.append(v)
    self.accept = not self.accept

  def add_collection(self, v_collection):
    for v in v_collection:
      self.add_item(v)
```

Comment

The solution employing **inheritance** only partially meets the requirements, as the object also responds to all the methods in the superclass `list`.
By using the methods inherited from the `list` class, any value can be stored, bypassing the specification that requires only the elements with odd indexes to be stored.

```
obj.append()
obj.extend()
```

If we intend to restrict the use of methods inherited from the list, we must override them. This necessitates reviewing and potentially modifying an extensive list of methods.

The solution utilizing **composition** grants us the ability to selectively choose which methods to expose.

Comment

```python
def add_item(self, v):
  if self.accept:
    self.values.append(v)
  self.accept = not self.accept
```

We can delegate the process of **changing the state** (accepting or rejecting the item) to a function.

We'll explore more general solutions as the logic to be implemented becomes more complex.