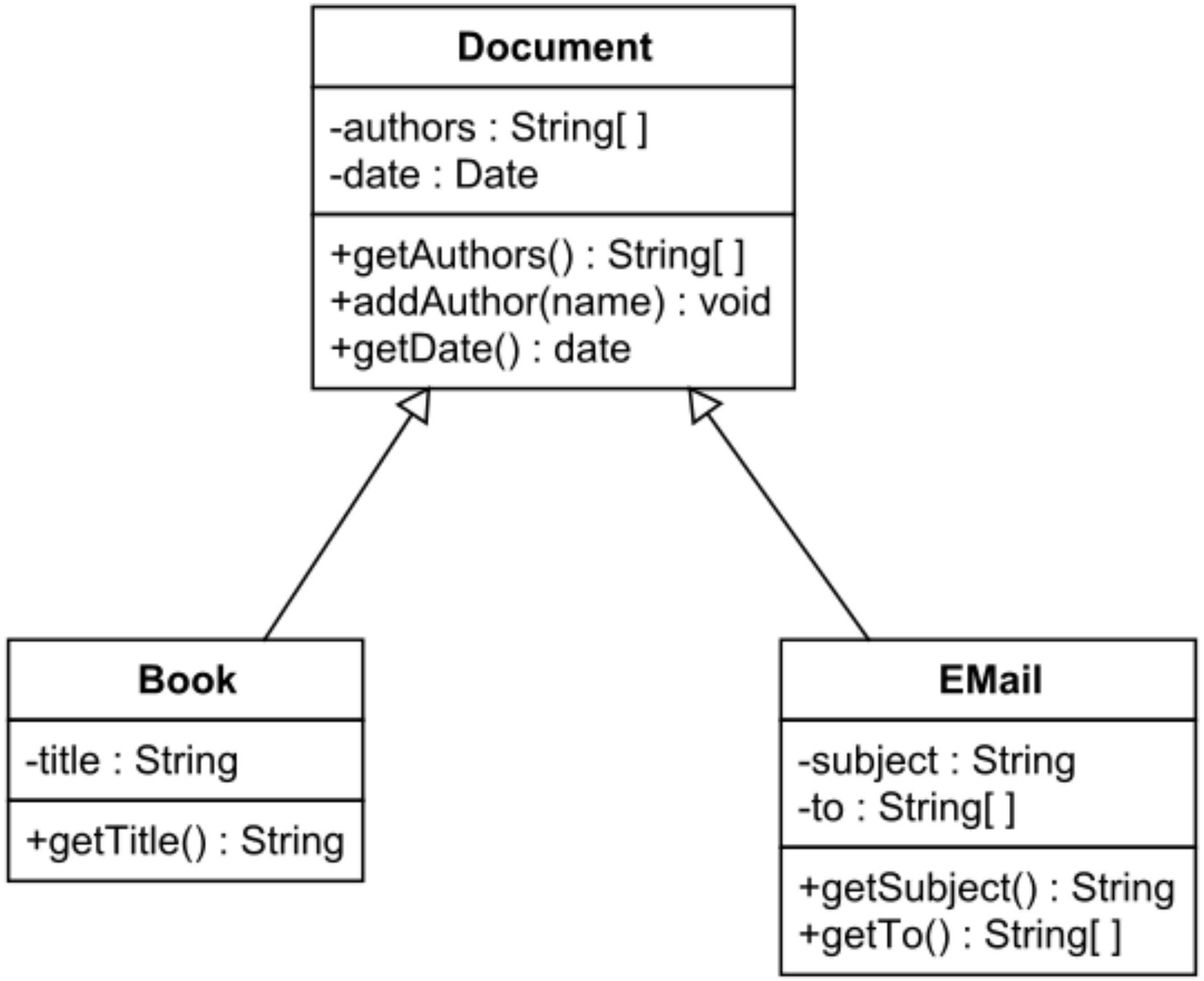
Introduction to OOP (2)

Inheritance

Inheritance is the mechanism of basing a class upon another class (or another object), retaining similar implementation.

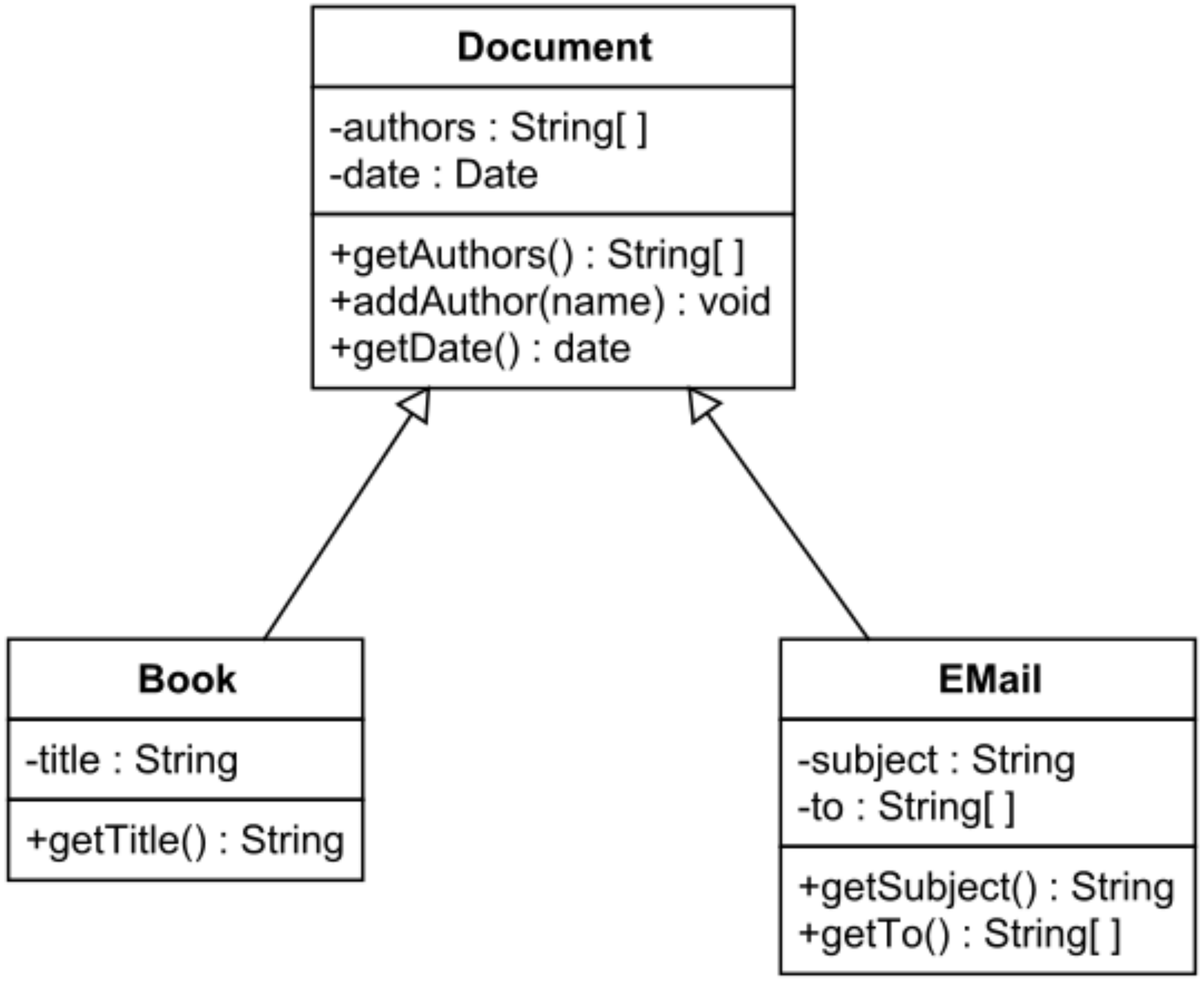
- Superclass, base classes, or parent classes
- Subclasses, derived classes, or child classes



Inheritance: Specialization

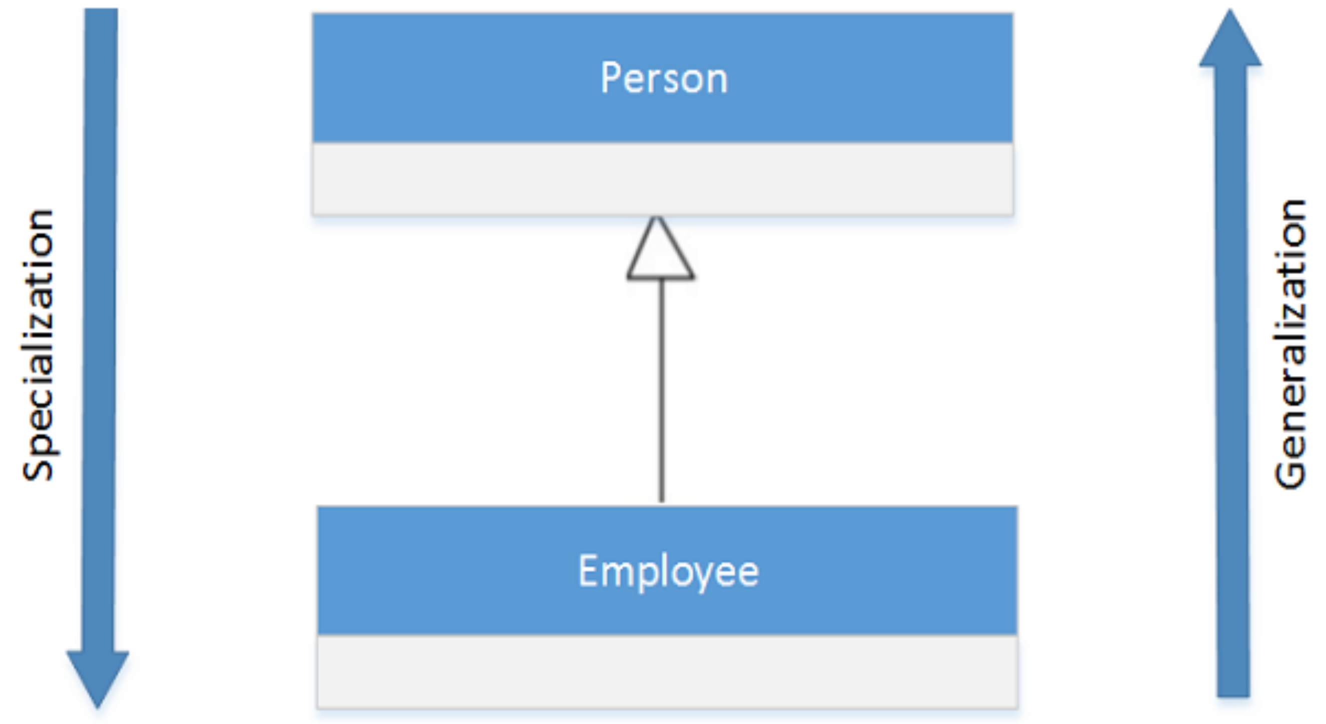
A class **extends** another class. The `Book` class and the mail class extend the `Document` class.

The Book and Email classes **inherit the fields and methods** of the Document class (possibly modifying the methods), but might **add** additional **fields and methods**.



Inheritance: Specialization

A class **extends** another class.



The Employee class extends the Person class (Inheritance)

Inheritance - (*Python implementation*)

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

```
class MySubClass1(object):
    pass

class MySubClass2:
    pass
```

A **subclass** (es. `MySubClass1`) is derived from its **parent class** (`object`). The subclass **extends** the parent class.

Add new behavior to existing class.

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

```
class MyClass:
    def __init__(self):
        # doSomething()
    def f1(self):
        # do ...

class MySubClass(MyClass):
    # it uses the __init__ and f1 method of the superclass
    def f2():
        # do ...
```

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):
        # do ...

class MySubClass(MyClass):
    # it uses the __init__  method of the superclass
    def f1(self): # redefine the method
        # do ...
```

Change behavior to existing class.

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

```python
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()
```

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**.

Change behavior to existing class.

```
class MyClass:
    def f1(self):
        # do_1()
        # do_2()
class MySubClass(MyClass):
    def f1(self): # redefine the method
        super().f1()
        # do_3()
```

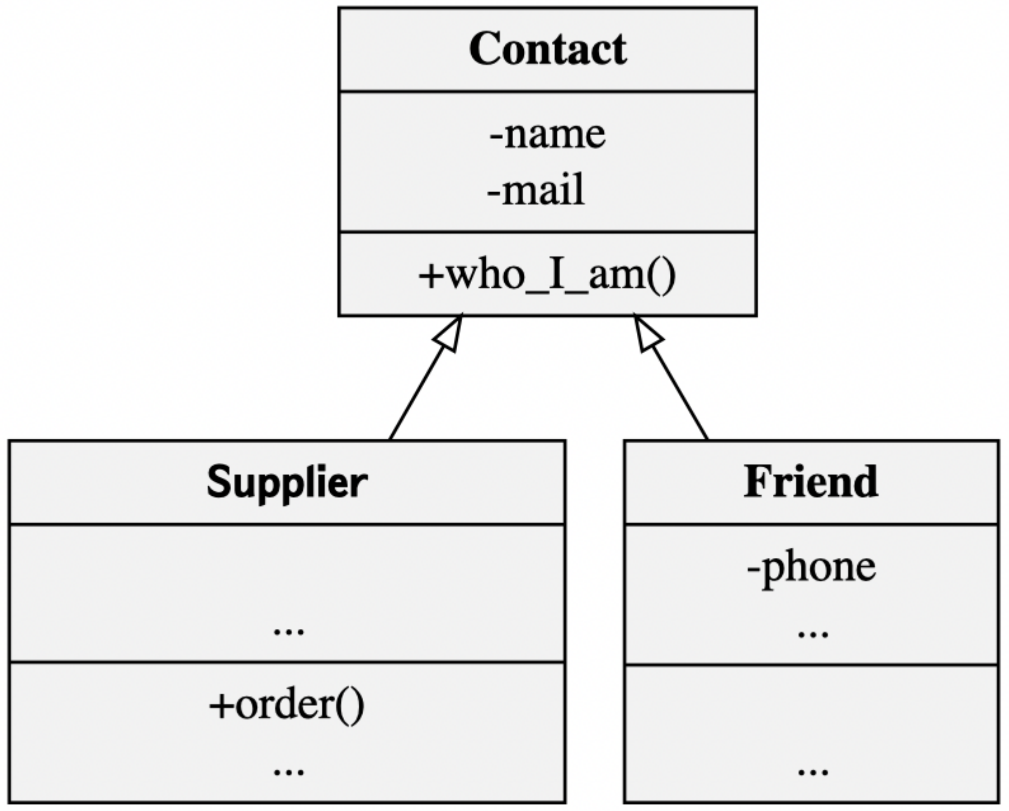
The `super()` function returns an instance of the parent class, allowing us to call the parent method directly. A `super()` call can be made inside any method. All methods can be modified via overriding and calls to `super()`.

Example (`inheritance_01`)

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

---

[1] Trivial approach: email address must contain `@`, and other characters before and after `@`. Use the String `split()` method. For a complete solution you can use regular expression `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.

Liskov substitution principle

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

Equivalent form:
"Functions that use pointers of references to base classes must be able to use objects of derived classes without knowing it." (Martin, 1996)

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

```
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.

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

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

Square and Rectangle, a More Subtle Violation.

```python
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, a):
        self._a=a

    @property
    def b(self):
        return self._b
    @b.setter
    def b(self, b):
        self._b = b

class Square (Rectangle):
      #...
```

- A Square does not need both `a` and `b` attributes
- Square will inherit the `a` and `b` setters. These functions are inappropriate for a Square, since the width and height of a square are identical
- You could override the setters so that if someone modifies `a`, `b` is modified in the same way, and vice versa. Thus, the Square object will remain a mathematically proper square.

Square and Rectangle, a More Subtle Violation.

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

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

    @a.setter
    def a(self, a):
        self._set_a(a)

    def _get_a(self):
        return self._a

    def _set_a(self, a):
        self._a = a

    # the same for b
```

Note

When you overload the property in a child class you must explicitely provide both getter and setters.

If you want to make the getter and/or setter overloadable without having to redeclare the property, you have to define `_get_()` and `_set_()` methods and make your getters and setters delegate to these methods.

Square and Rectangle, a More Subtle Violation.

```python
class Rectangle:
    def __init__(self, a, b):
        self._a = a
        self._b = b

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

    @a.setter
    def a(self, a):
        self._set_a(a)

    def _get_a(self):
        return self._a

    def _set_a(self, a):
        self._a = a

    # the same for b
```

```python
class Square(Rectangle):
    def __init__(self, a, b=None):
        self._a = a
        self._b = a

    def _set_a(self, a):
        self._a = a
        self._b = a

    def _set_b(self, b):
        self._a = b
        self._b = b
```

Now the Square object will remain a mathematically proper square.

The model is now self-consistent, but it is not consistent with all its uses!
(code: `intro_oop/lsp_1.py`)

What happens?

```python
def f(s):
    s.a = 10
    s.b = 3
    print(s.a * s.b == 30)
```

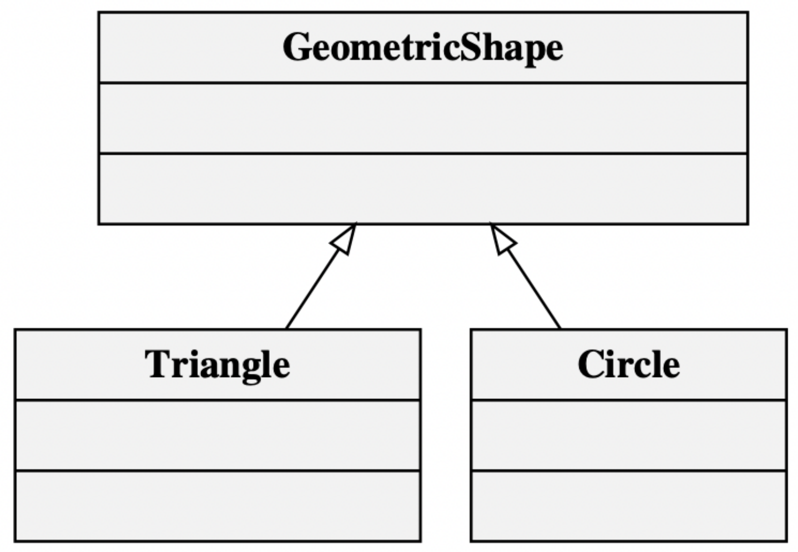
A model, viewed in isolation, can not be meaningfully validated. The validity of a model can only be expressed in terms of its clients.

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

The behavior of a `Square` object is not consistent with the behavior of a `Rectangle` object.

Possible solution: make the objects immutable.

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 classes that 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 allows us to indicate which methods must be overwritten.

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
        print("JUMP!")

class Mouse(Character):
    def jump(self):
        super().jump() #This part is implemented in Abstract Class
        print("A mouse can jump!")

# p1 = Character('Mickey Mouse')
p1 = Mouse('Mickey Mouse')
p1.jump()
#JUMP!
#A mouse can jump!
```

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.

```python
class Character(ABC):
    #...
    @abstractmethod
    def jump(self): #you are forced to redefine this method
        print("JUMP!")

class Mouse(Character):
    def jump(self):
        super().jump() #This part is implemented in Abstract Class
        print("A mouse can jump!")
    #...
```

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 has been said 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)

```
class A:
    def __init__(self):
        self.a=0
        self.z=0

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

(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
```

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** or **composition**.
- In some situations *inheritance* is the best solution: **the Orange class extends the Fruit class**.
- In other situations *composition* models our system better: **a car is composed of an engine, four wheels**, and so on[2].

Composition implements a has-a relationship, in contrast to the is-a relationship of subclassing.

[2] From a syntactic point of view it is possible to extend (using inheritance) the class Engine adding attributes that represent wheels, seats, steering wheel, and so on. We can think a car as if it were *an engine with wheels and seats*, but this is a bad modeling of reality. It is much more consistent to think of the car as the *union of the various parts that compose it*.

Exercise

Write a class `Car` using **inheritance** (a `Car` **is** an `Engine` that has also `seats` and `wheels`) and **composition** (a `Car` is a **composition of** `engine`, `seats` and `wheels`).

(`intro_oop/car_inheritance`; `intro_oop/car_composition`)

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