Introduction to Python

Session 02

Introduction to Object Oriented Programming (OOP)

Encapsulation
Inheritance

Encapsulation in OOP

Encapsulation:

A method for **restricting access** to object's components (variables or methods).

- Used to hide the values or state of a structure data object inside a class.

- Protects object integrity by controlling how data is accessed and modified

- **Private:** Accessible only within the **class** itself
- **Protected**: Accessible within the **class** and its **subclasses**
- **Public:** Accessible from **anywhere** in the code

upf.

Encapsulation in OOP

Encapsulation:

In Python, everything is public !!!

But the is a naming convention, not enforced restrictions to indicate it:

- **Private:** `__member_name` 2 underscores to make it private

  Name mangling: Makes it harder to accidentally access from outside the class. Automatically renamed to `_ClassName__member_name`

- **Protected**: `_member_name`.

- **Public:** By default, all is public.

upf.

Encapsulation in OOP

Encapsulation examples:

- **Public:** By default, everything is is public.

```
class Cup:

    def __init__(self):
        self.content = None

    def fill(self, beverage):
        self.content = beverage

    def get_beverage(self):
        return self.content
```

Public attribute

Public method

Public method

upf.

Encapsulation in OOP

Encapsulation examples:

- **Public:** By default, everything is is public.

```
>>> cup_instance = Cup()

>>> cup_instance.fill("Coffee")

>>> print(cup_instance.get_beverage())
Coffee

>>> print(cup_instance.content)
Coffee
```

upf.

Encapsulation in OOP

Encapsulation examples:

- **Private:** Nobody should be a able to access it from outside the class.

```
class Cup:

    def __init__(self):
        self.__content = None

    def fill(self, beverage):
        self.__content = beverage

    def get_beverage(self):
        return self.__content
```

Private attribute

we don't want the user to use the attributes, only the methods

Public method

Public method

private means that the class can only modify itself.

inside class we have access to everything
but if we import it, we only have access to whats public, the methods in this case

upf.

Encapsulation in OOP

Encapsulation examples:

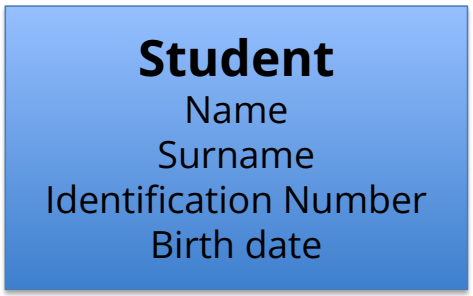
- **Private: __**

```
>>> cup_instance = Cup()

>>> cup_instance.fill("Coffee")

>>> print(cup_instance.__content)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Cup instance has no attribute '__content'

>>> print(cup_instance.get_beverage())
Coffee
```

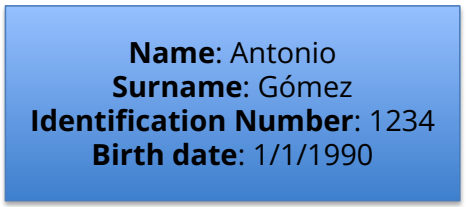
upf.

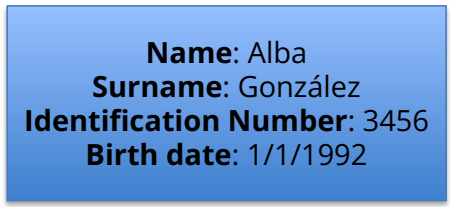
Object Oriented Programming

Class: Defines the structure: attributes and methods



Instances: Specific realization of any class. Objects that exist in a given program execution





upf.

Class attributes

Attributes (data): Variables that define the state of a class instance.

- Defined inside the constructor method `__init __`
- `self.`

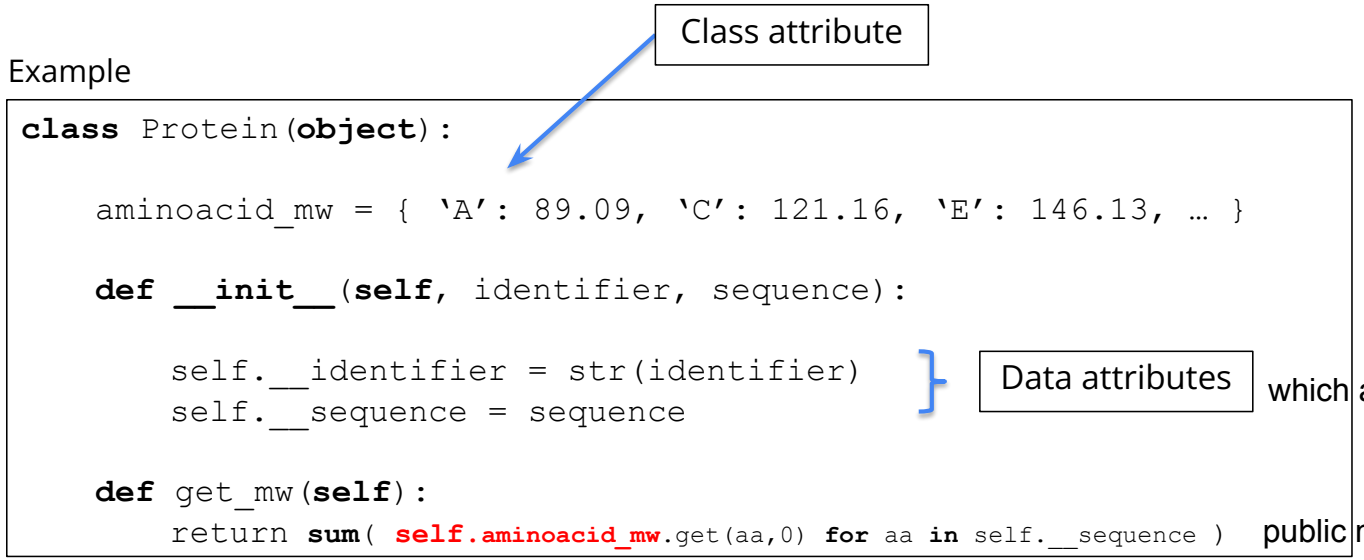
Class attributes: Variables that define “class-level” **constants**.

- Same variable **shared** by all instances.
- Defined after the class statement outside the constructor `__init__`
- Defined without `self.`
- Accessed both inside the instance using `self.` and as a property of the class.

upf.

Class attributes

Example



this is a list comprehension, create a list with aa that comes from the iterator sequence, and put them in a list

- **Reading**: All instances can access the same class attribute
- **Modifying** via the **class**: Changes are visible to all instances
- **Modifying** via an **instance**: Creates a new instance attribute that shadows the class attribute (only for that instance) When **mutable** (list, dict, set), modifying the object itself (not reassigning) affects all instances:

upf.

Class attributes


```
>>> p1 = Protein(1, "AC")
>>> p2 = Protein(2, "AE")

>>> p1.aminoacid_mw
{'A': 89.09, 'C': 121.16, 'E': 146.13}

>>> Protein.aminoacid_mw
{'A': 89.09, 'C': 121.16, 'E': 146.13}

>>> p1.get_mw()
210.25

>>> p2.get_mw()
235.22

>>> Protein.aminoacid_mw["A"] = 145.23
>>> p2.get_mw()
291.36
```

Call to the constructor

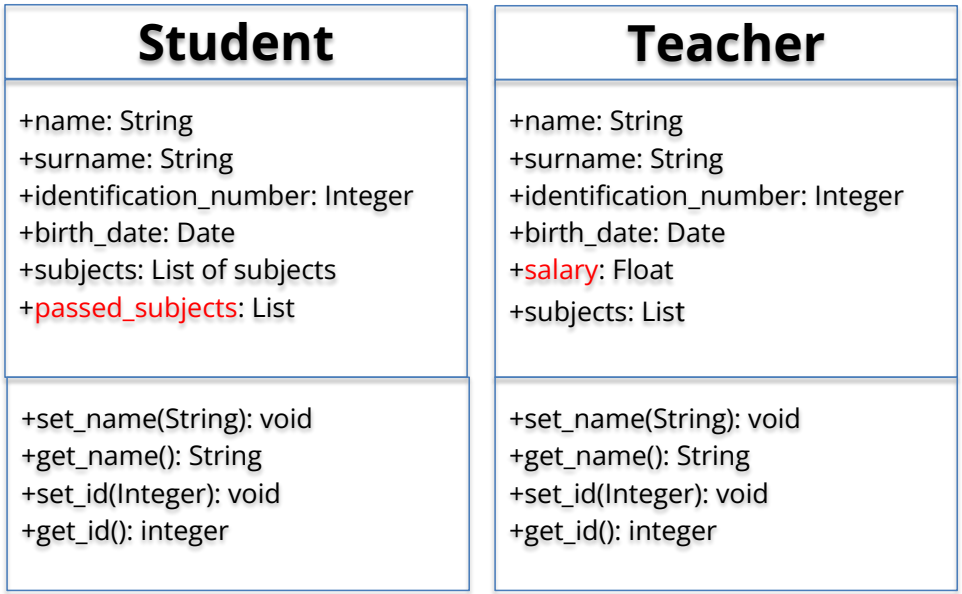
Accessed as an instance property

Accessed as a class property

Shared property between all instances.

upf.

Inheritance



here we have two classes, and most of the fields are shared.

Inheritance

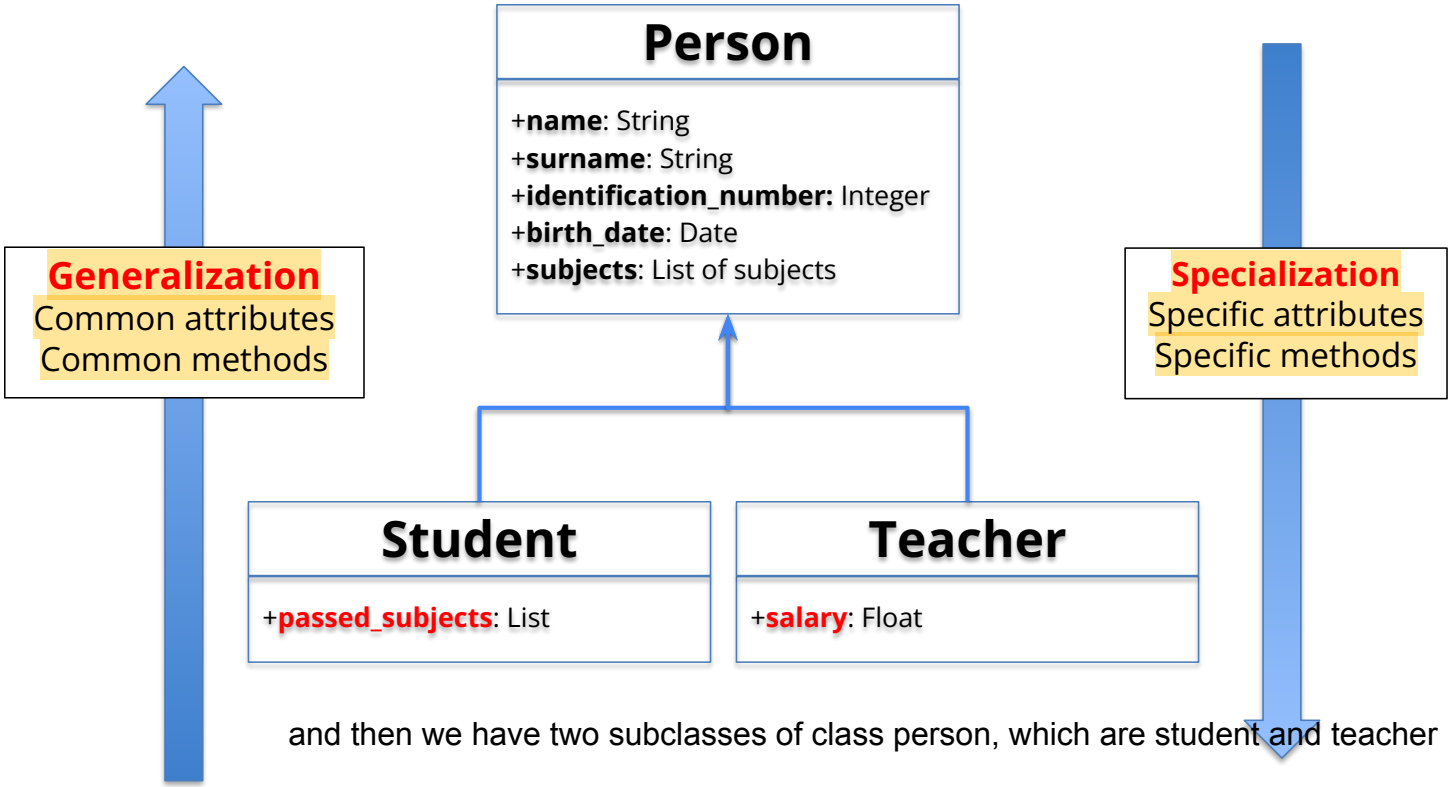
Student
+**name**: String +**surname**: String +**identification_number:** Integer +**birth_date**: Date +**subjects**: List of subjects +**passed_subjects**: List
+set_name(String): void +get_name(): String +set_id(Integer): void +get_id(): integer

Teacher
+**name**: String +**surname**: String +**identification_number**: Integer +**birth_date**: Date +**salary**: Float +**subjects**: List
+set_name(String): void +get_name(): String +set_id(Integer): void +get_id(): integer

upf.

Inheritance

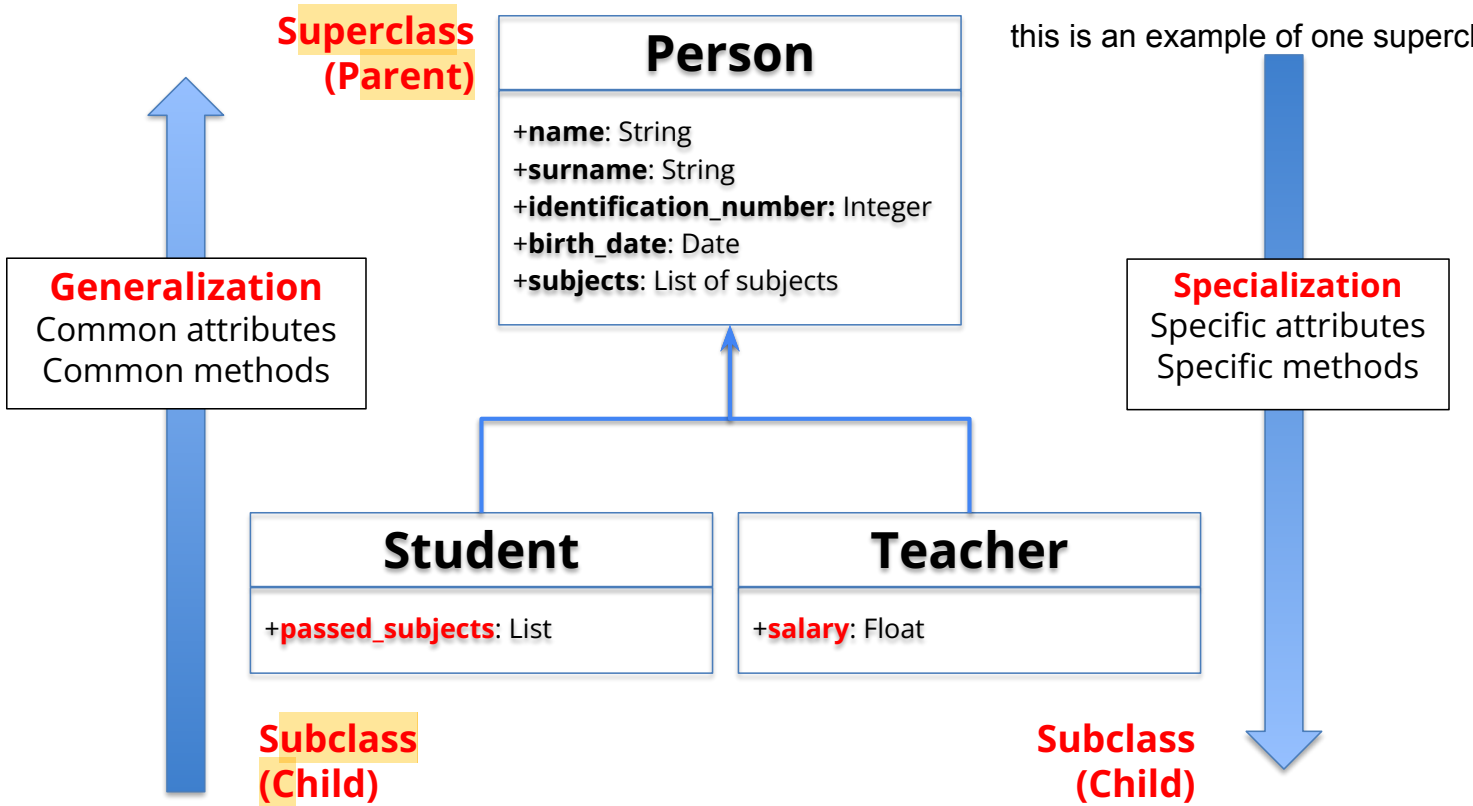
we might have a genral class, and the attributes are shared between student and teacher



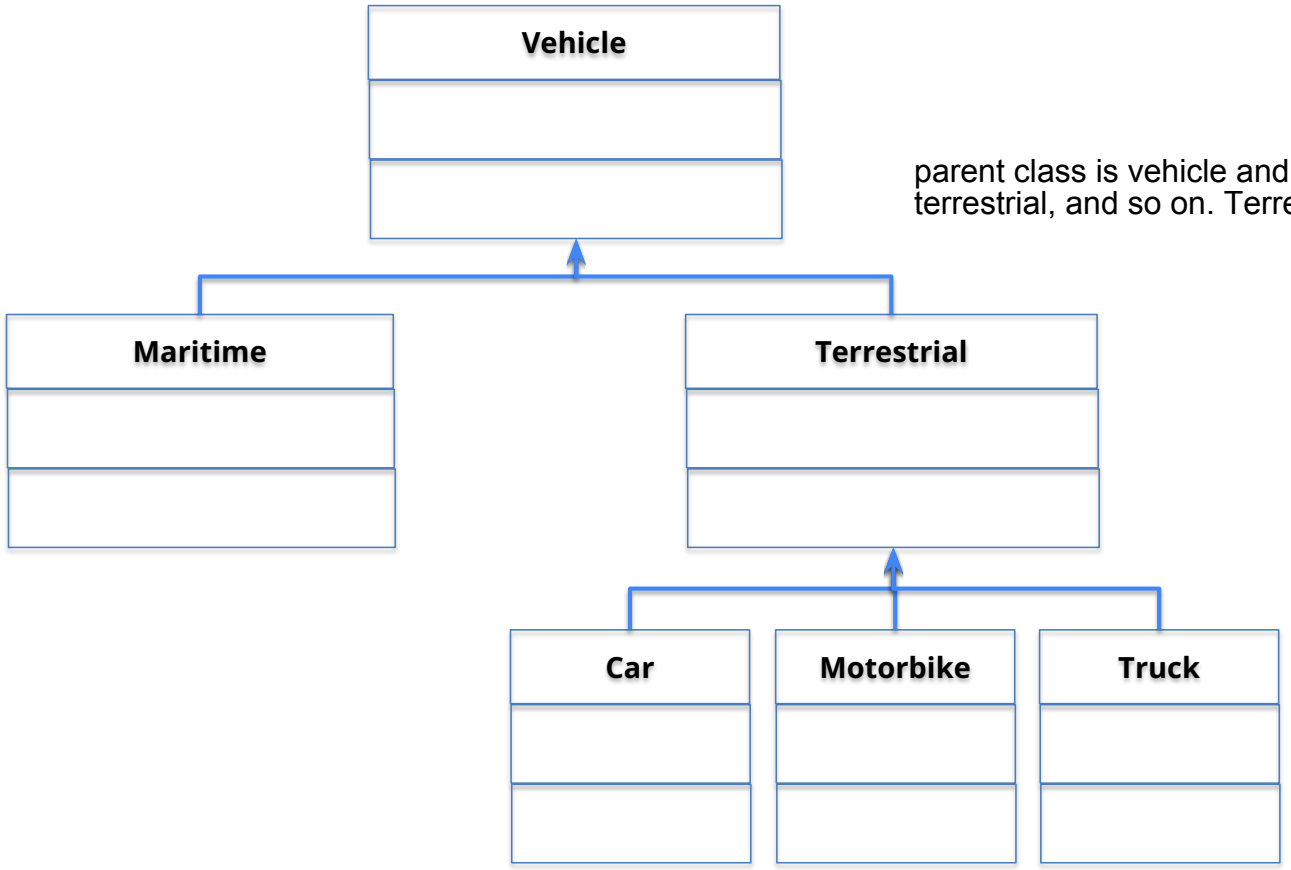
and then we have two subclasses of class person, which are student and teacher

upf.

Inheritance



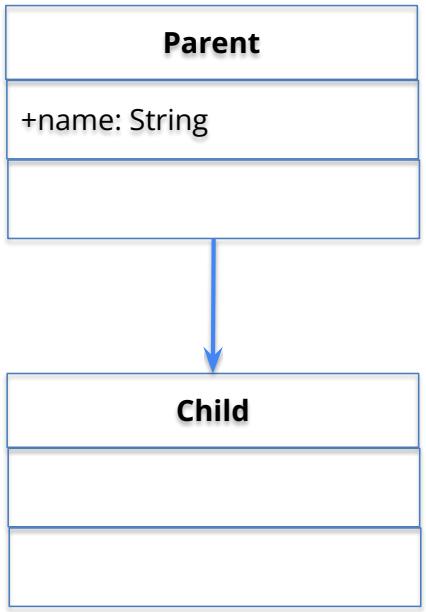
Inheritance



parent class is vehicle and we specialize itno maritime and terrestrial, and so on. Terrestrial is a superclass of car.

upf.

Implementing Inheritance




```
class Parent():

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

class Child(Parent):
    pass
```

Superclass

Create an instance of class Parent:

```
parent_instance = Parent("ABC")
```

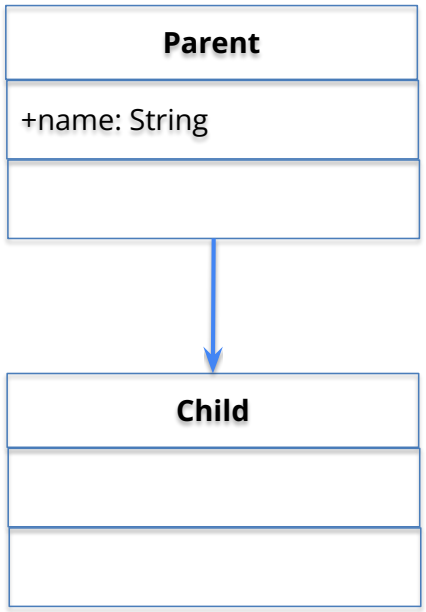
Create an instance of class Child:

```
child_instance = Child("TTT")
```

upf.

Implementing Inheritance

Childs inherit ALL attributes and methods from its superclass !!




```
class Parent():

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

class Child(Parent):
    pass
```

Superclass

Create an instance of class Parent:

```
parent_instance = Parent("ABC")
```

Create an instance of class Child:

```
child_instance = Child("TTT")
```

upf.

Implementing Inheritance

In this example, the Child class **inherits** the method print_name from its superclass

```
class Parent():

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

    def print_name(self):
        print("I am the parent and my name is %s" %self.name)

class Child(Parent):
    pass
```

```
>>> parent_instance = Parent("ABC")
>>> parent_instance.print_name()
I am the parent and my name is ABC

>>> child_instance = Child("TTT")
>>> child_instance.print_name()
I am the parent and my name is TTT
```

upf.

Overriding methods and attributes

In this example, the Child class **overrides** the method print_name

```
class Parent():

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

    def print_name(self):
        print("I am the parent and my name is %s" %self.name)

class Child(Parent):

    def print_name(self):
        print("I am the child and my name is %s" %self.name)
```

```
>>> parent_instance = Parent("ABC")
>>> parent_instance.print_name()
I am the parent and my name is ABC

>>> child_instance = Child("TTT")
>>> child_instance.print_name()
I am the child and my name is TTT
```

upf.

Overriding methods and attributes

Overriding: Re-implementing a method for a Child class:

- You want the Child to behave differently.
- Replace a functionality
- Add new functionality to an existing method.

Overriding the constructor method:

```
class Parent():

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


class Child(Parent):

    def __init__(self, name, age):

        self.age = age
        super().__init__(name)
```

upf.

Overriding methods and attributes

Overriding: Re-implementing a method for a Child class:

- You want the Child to behave differently.
- Replace a functionality
- Add new functionality to an existing method.

Overriding the constructor method:

```
class Parent(object):

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

class Child(Parent):

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

Initializer of the superclass

Arguments for the initializer

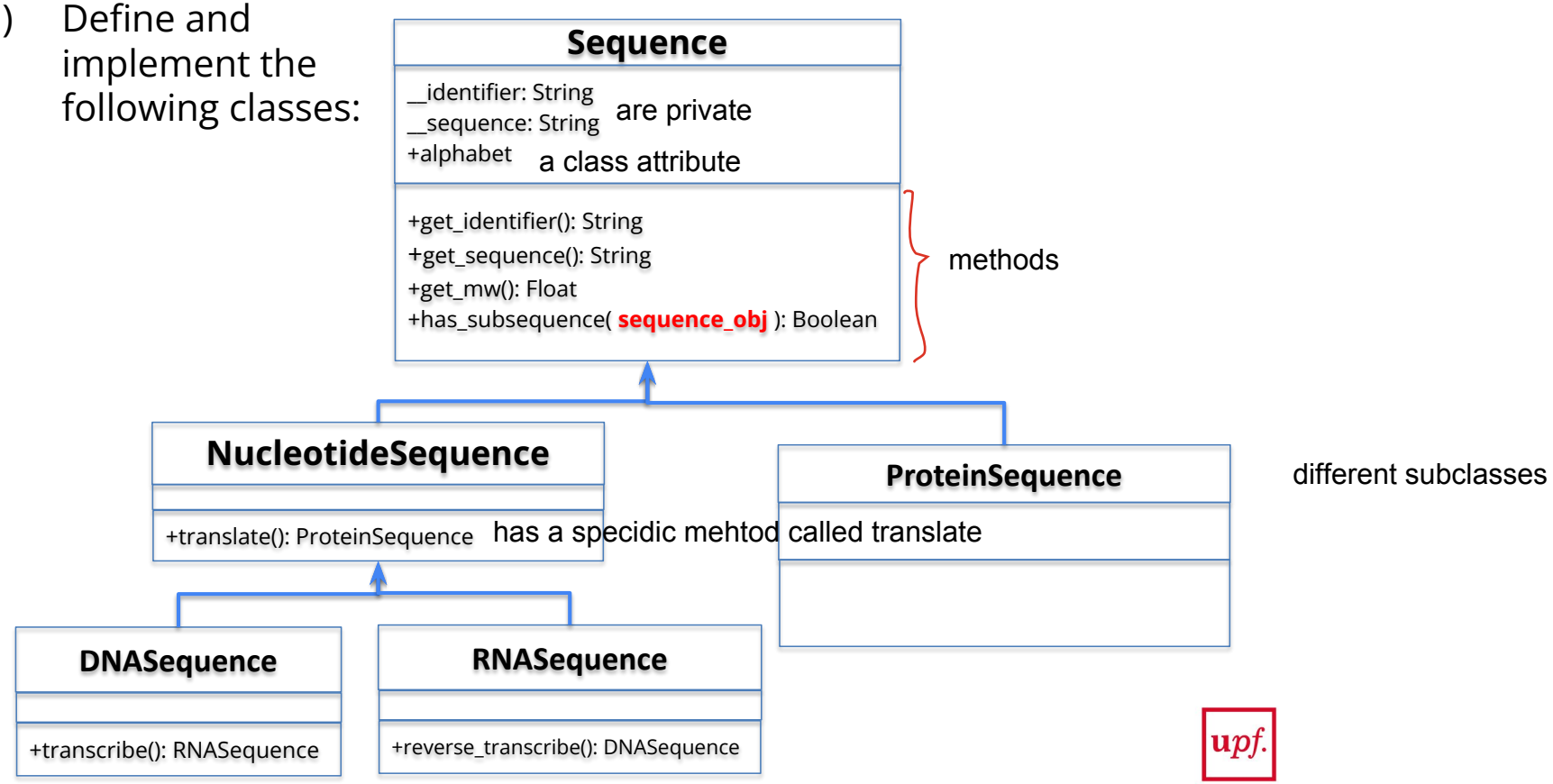
Returns the parent class

upf.

Exercises. Session 08

Create a python script called **NIE_exercise_block2_part2.py** with:

1) Define and implement the following classes:



Exercises. Session 08

Specifications:

1. `alphabet` must be a **class attribute** that specifies the possible alphabet of the sequence.

2. Only ProteinSequence, DNASequence and RNASequence are instantiated.

3. When creating a new Sequence instance (ProteinSequence, DNASequence or RNASequence), it must check that the sequence is correct by checking in the alphabet. If not, **raise an exception** with the following statement, where *X* is the incorrect letter: `raise ValueError("Impossible to create instance: X not possible")`

upf.

Exercises. Session 08

Specifications:

4. You can find required data in a file called: `Sequence_dictionaries.py`

5. No need to take into account alternate ORF for traduction.

6. Biological accuracy will not penalize if reasonable.