IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

OBJECT ORIENTED PROGRAMMING (PYTHON)

1. WHAT IS OOP
2. CLASSES AND INSTANCES
3. INHERITANCE
4. ENCAPSULATION
5. POLYMORPHISM

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

WHAT IS OOP?

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

INTRODUCTION

Computer programming paradigm

- Everything is an object
- Useful for large program, actively updated, shared
- Programmer defined type
- We call this new custom type : class
- Class have attributes and functions

Python is an OOP language

- Python “Think Python” (Chapter 15+)

Un example

The class « Cat »

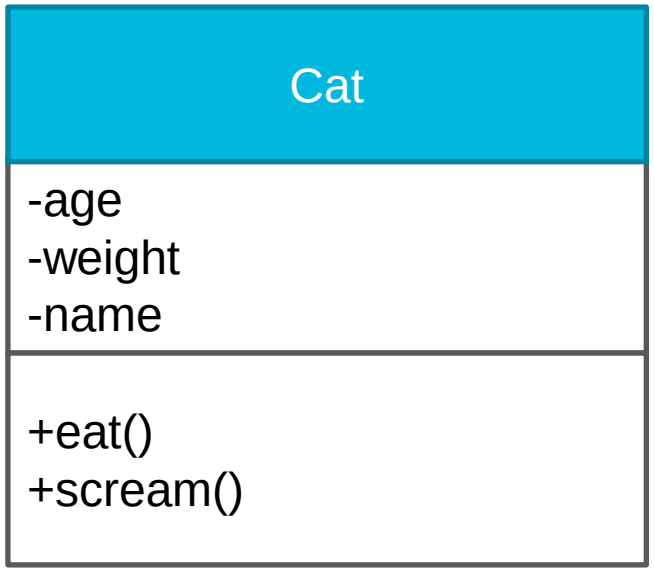
- The class: all the cats
- An instance: a cat

- Attributes:
  - age
  - weight
  - name
  - …
- Function
  - eat()
  - scream()

The Unified Modeling Language (UML)

- Standard way to visualize a system

A class:

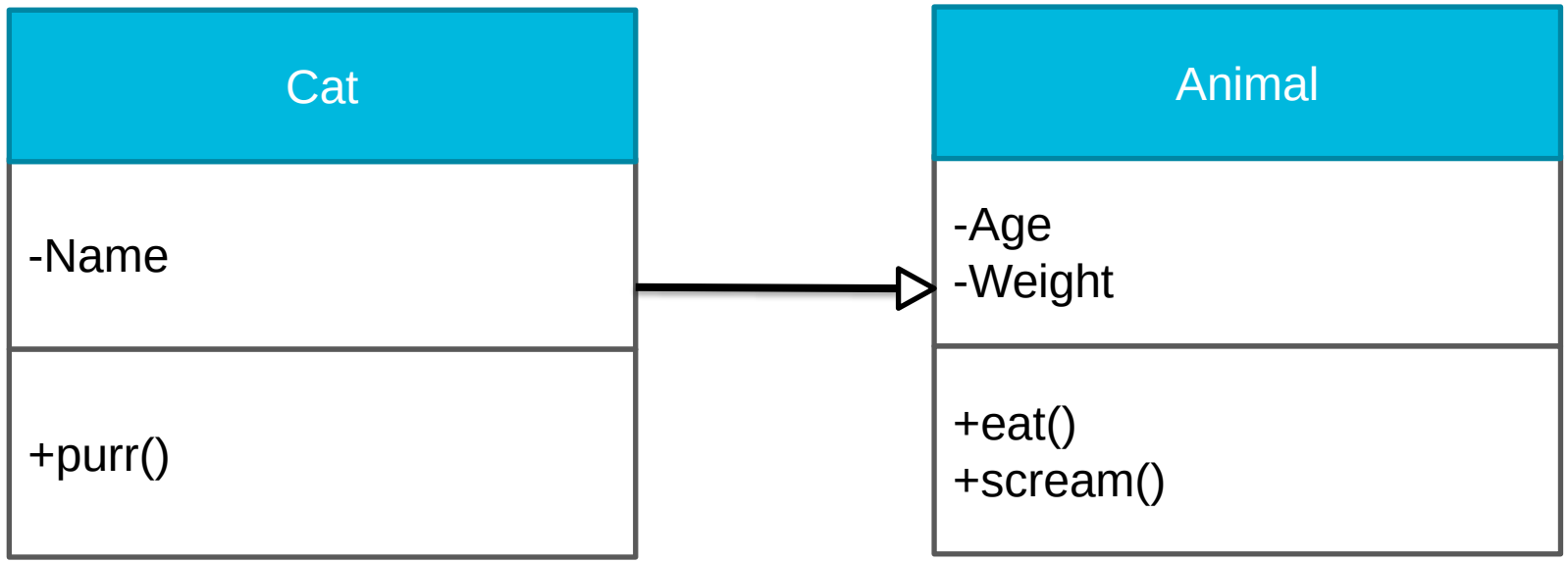


All the cat are animals

- A cat is an Animal
- All animals are not cats

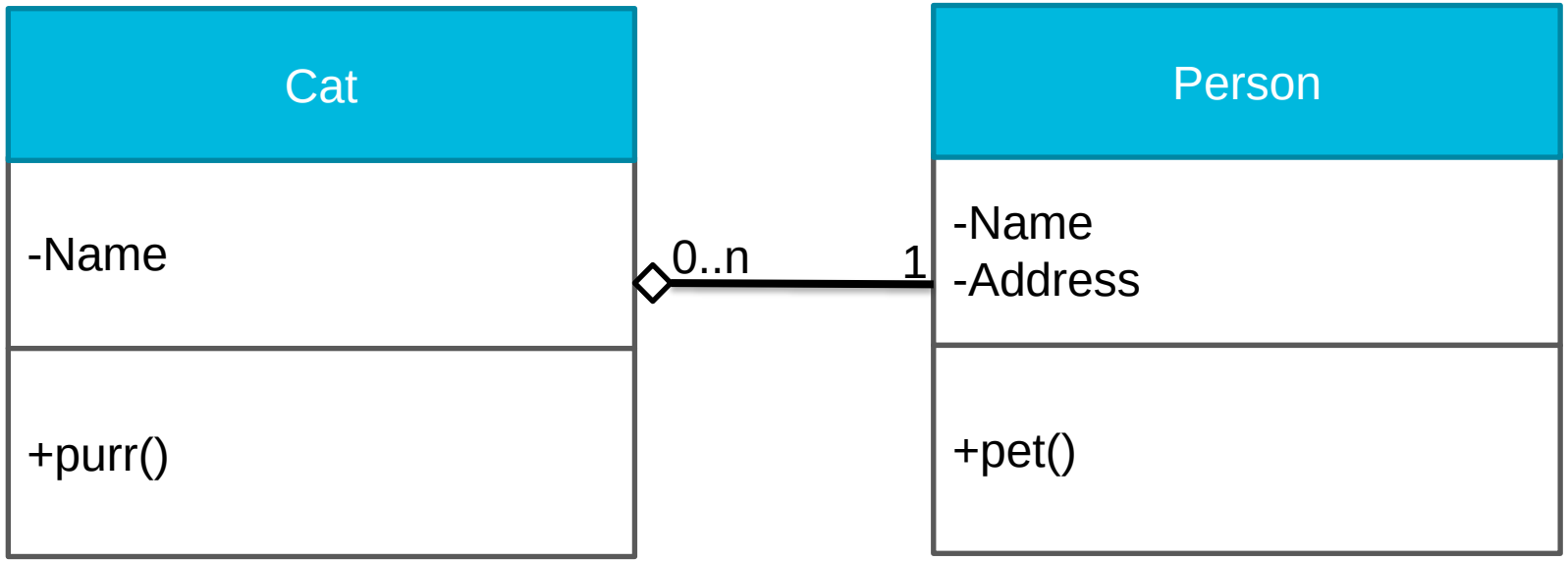
We say that Cat « inherit » from Animal

A cat pocess all the attributes and functions of Animal

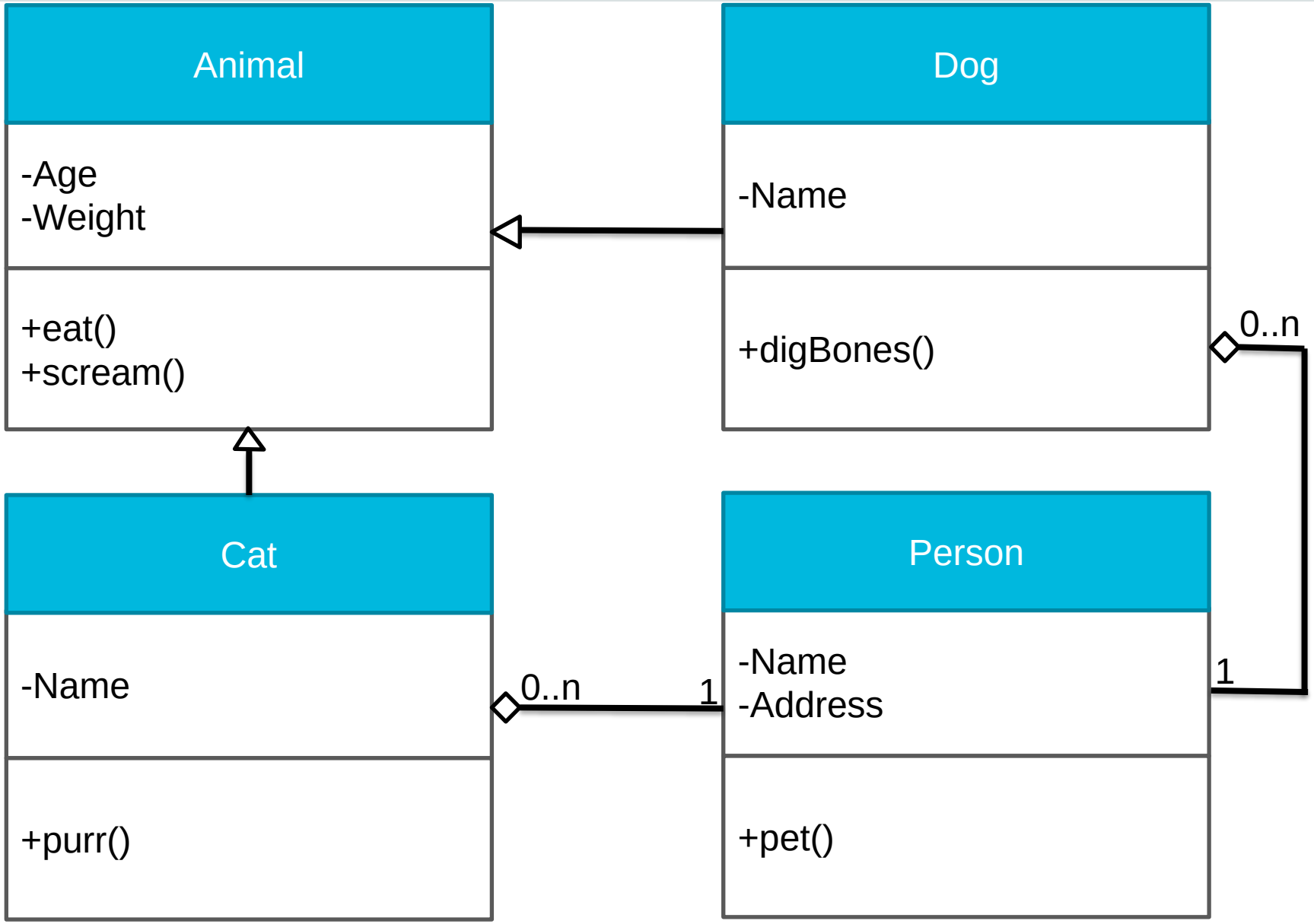


All cat has an Owner

- The class cat has a relation of agregation with the class owner



Full example



Exercice

There is two type of cars, A and B. The cars A have 4 wheels but the cars B have 6 wheels. All the cars have an engine that can be or electric or thermal.

- Write the UML class diagram corresponding to this example.

Anaconda => Spyder

```
class Cat:
...
```

Felix=Cat()

Felix.eat() // from Animal
Felix.purr() // from Cat
Felix.name=“Felix”
print(Felix.name) #Felix

Initilization

__init__

The method called by Cat()

```
class Cat:
    def __init__(self, name):
        self.name=name

Felix=Cat("Felix")
print(Felix.name)
Print(Felix)
```

__str__

The method call when printing the object (print)

```
def __str__(self):
        return "i'm a Cat and my name is "+self.name

print(Felix)
```

We can define function for a given object or for the whole class

- For an object:

```
def purr(self):
        return 'ronronronron’
```

- For the class

```
def getTheBestBreed():
        return "Maine Coon“
```

How do we call them?

- For an object: felix.purr():
- For the class: Cat.getTheBestBreed()

Try: Felix.getTheBestBreed()

Inheritance

```
class Animal:
 def __init__(self, age):
        if(age>=0):
            self.age=age
        else:
            raise NameError('NegativeAge’)

class Cat(Animal):
def __init__(self, name, age):
        Animal.__init__(self,age)
        self.name=name
```

Exercise

1. Create a class Point with attribute x and y corresponding to its coordinate. Write the code for the function “__str__” and “__init__”. Test these methods.
2. Create a function “cartesianDistance” to compute the cartesian distance between two “Point”.
3. Test your function over the two points (0,5) (-1,9)
4. Create a sub-class of Point named City. The cities have a name and a number of inhabitant. Write the code for the function “__str__” and “__init__”.
5. Create a function to add a number of inhabitant in the City (+500, -200 for example)

Let's define this new class “Owner”

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

- Now we define the Owner of the Cat :

```
class Cat(Animal) :
def __init__(self,nam,age,person):
        Animal.__init__(self, age)
        self.name=nam
        self.owner=person
```

Exercice

1. Create a class “Country”. This class has a number of inhabitant and a capital city (which is… a City!) and a table of other cities.
2. Create a function to add a City to the table of cities of a Country
3. Create a function to remove a City from the table of cities of a Country

4. Add a function “isACapitalCity” to the class City, that return True if the City is the capital of a country already created, False otherwise. Test your method with the objects: France, Spain, Madrid, Paris, Rome, Nantes

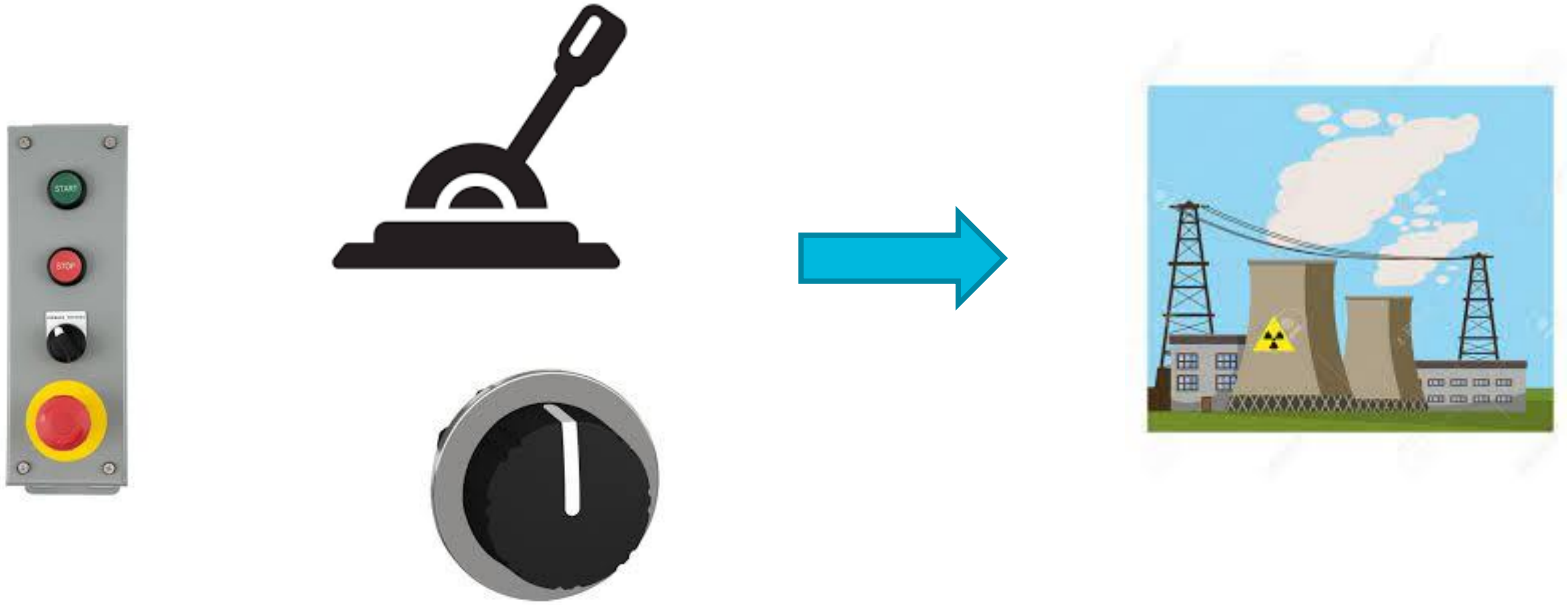
ENCAPSULATION

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Encapsulation

“Encapsulation refers to the building of the data with the methods that operate on the data”

- Hide the implementation for the user
- Protect against the violation of state and values of the system



IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Class attribute

It is also possible to define attribute for the whole class

```
class Cat(Animal):
    temper="bad"
    lastId=0 #useful to set a different id for each cat

 def __init__(self, name, age):
        self.id=Cat.lastId+1
        Cat.lastId+=1
```

What happen if the args “lastId” is changed out of the class?

```
Felix=Cat("Felix",5)
print(Felix.id) #1
Paul=Cat("Paul",4)
print(Paul.id) #2
```

What if we add :

```
Felix=Cat("Felix",5)
print(Felix.id) #1
Cat.lastId-=1
Paul=Cat("Paul",4)
print(Paul.id) #?
```

Encapsulation

Visibilty:

- Public: can access from everywhere
- Protected: can only access from sub-class
- Private: can access only from the class itself

Python does not provide a formal implementation of these visibilities

Public: name
Protected: _name
Private: __name (this one works (almost))

Back to the example

```
class Cat(Animal):
    temper="bad" #means that all cat are nasty
    __lastId=0 #useful to set a different id for each cat

 def __init__(self, name, age):
        self.id=Cat.__lastId+1
        Cat.__lastId+=1

print(Felix.id)
Cat.__lastId-=1 #fail
Paul=Cat("Paul",4)
print(Paul.id)
```

Copy

```
A=2
B=A
B+=2
Print(A,B)
#2 4
```

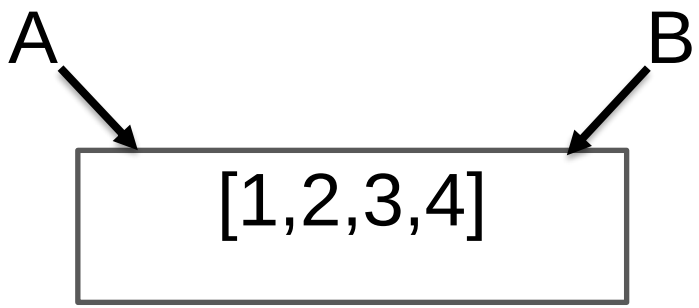
- On primitive type, python does a copy with the affectation

What about complex type?

Affectation

```
A=[1,2,3,4]
B=A
B.pop(0)
print(A,B)
#[2, 3, 4] [2, 3, 4]
```

Not a copy, just a “pointer”



Copy

```
Import copy
A=[Paul,2,3,4]
B=copy.copy(A)
B.pop(0)
print(A,B)
```

[<__main__.Cat object at 0x000001F247FEBAC0>, 2, 3, 4] [2, 3, 4]

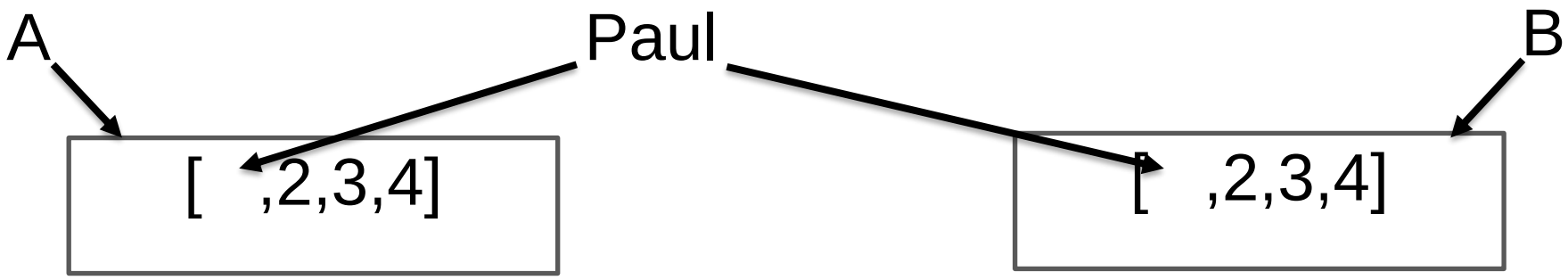
The method called here is __repr__(repr:debug, str: end users):

```
 def __repr__(self):
        return self.name+str(self.age)
```

[Paul4, 2, 3, 4] [2, 3, 4] #=> the copy worked and paul has not been removed in A!

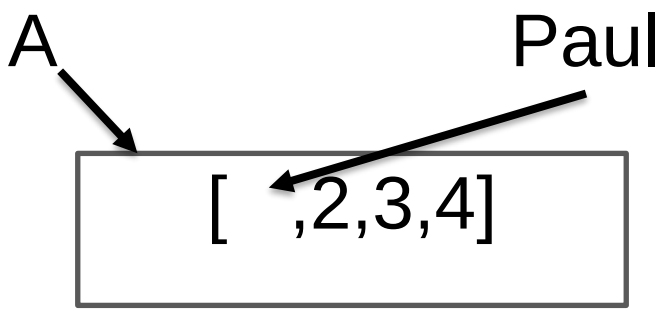
Ok but… did Paul was copied or not?

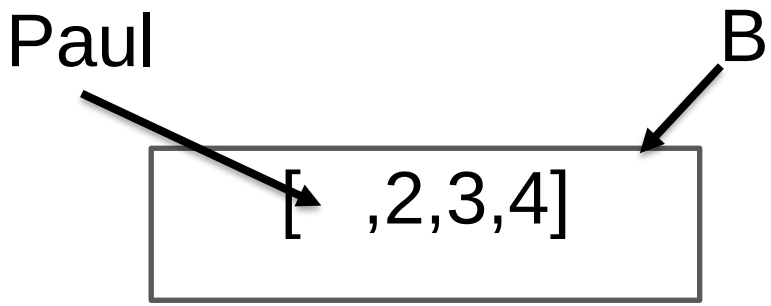
```
A=[Paul,2,3,4]
B=copy.copy(A)
B[0].age=99
print(A,B)
#[Paul99, 2, 3, 4] [Paul99, 2, 3, 4]
```



What if we want also Paul to be “copied”? => deepcopy

```
A=[Paul,2,3,4]
B=copy.deepcopy(A)
B[0].age=99
print(A,B)
#[Paul4, 2, 3, 4] [Paul99, 2, 3, 4] Paul is duplicated… id=same
```

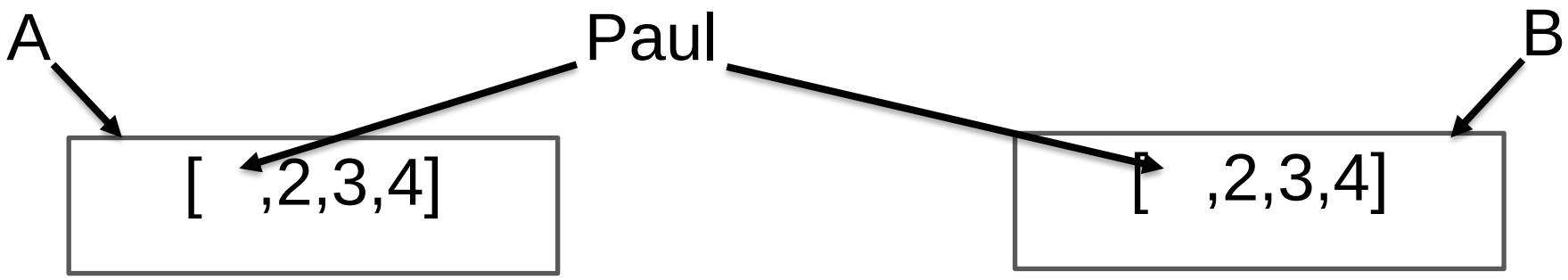




So should you use =,copy, or deepcopy?

- Depends on the situation!

In this example, deepcopy create inconsistency with two cats having the same id.
But for example, in a Local Search, when you want to test a neighbor, you don't want to modify the original solution, so a deepcopy can be useful.



Exercise

1. Add an id to the cities, such that all the cities have a different one when created
2. Create a class function areTheSameCities(A,B) that return True if the cities are the same, False otherwise. Test your method with the previously created cities (Paris, Nantes, Madrid).
3. Create a class function getTheBestCity(votes) returning the best city from votes. The entry “votes” is a table of “City”. Test your function with the following votes: [Madrid,Nantes,Madrid,Nantes,Nantes,Nantes,Paris,Paris]
4. Thanks to these votes, Nantes is now the new capital city of France! Change the new capital of France. Check if Paris.isACapitalCity. What do you propose to solve this problem?

POLYMORPHISM

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Polymorphism: having several forms

- In programming: the same function name, but a different behavior, depending on the parameters.
- Main polymorphism in python:
  - Two functions with the same name, but in different class
  - Two functions with the same name, same parameters, but in sub-classes
  - Two functions with the same name, in the same class, but with different parameters

- Two functions with the same name, but in different class

Example: __str__
If we print a Cat, the __str__ of Cat is called
If we print a City, the __str__ of City is called

- Two functions with the same name, same parameters, but in sub-classes

Paul is a Cat but is also an Animal. Which __str__ is called?
The most specific is called, the __str__ of Cat is called

- Two functions with the same name, in the same class, but with different parameters

If we want to create a cat, but we don’t know the age, how can we do?

- We can give “default” value to an argument.

```
class Cat(Animal):
    def __init__(self, age, name="john doe"):
```

- If only one argument is passed to __init__, the parameter “name” is set to “john doe”.
- This imply that the args with a default value must be in the last positions in the declaration of the function

What if we don’t know the number of arguments? Like “print” for example

```
 def addCatFriend(self,*arg):
        for friend in arg:
            self.friends.append(friend)

Paul.addCatFriend(Paul,Felix)
```

*arg is an infinite number of arguments (might be zero)

- Must be in last position! (after the arguments with default values)

Exercise

1. We consider two new classes: Person and Mayor. A Person has a name and an address. If the address is not known, the address is “unknown”. A Mayor is a Person and must be associated with a City.
2. Create a function isAMayor() for the class Person. Ensure that this function will return True if the Person is a Mayor, False otherwise.
3. Add in the class City a list of inhabitant. Create a function to add several Person to this list.
4. Create a function for the class Person called getDistanceFromAMayor(person) that return the distance between the two cities of the two mayors, if and only if they are both Mayor. Otherwise, it should return False.