Object-Oriented Programming in Python

Gloria Beraldo (gloria.beraldo@unipd.it)
Department of Information Engineering, University of Padova

Topics:

- Get started with OOP in Python
- Operators Overloading
- Parameters passing
- Cloning
- Polymorphism
- Inheritance

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

What is object-oriented programming?

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic.

An object can be defined as a data field that has unique **attributes** and **behaviors**.

Example:

Monkey

Attributes: name, age, colors, height, weight, etc.

Behaviors: eating, moving, manipulating, sleep, etc.

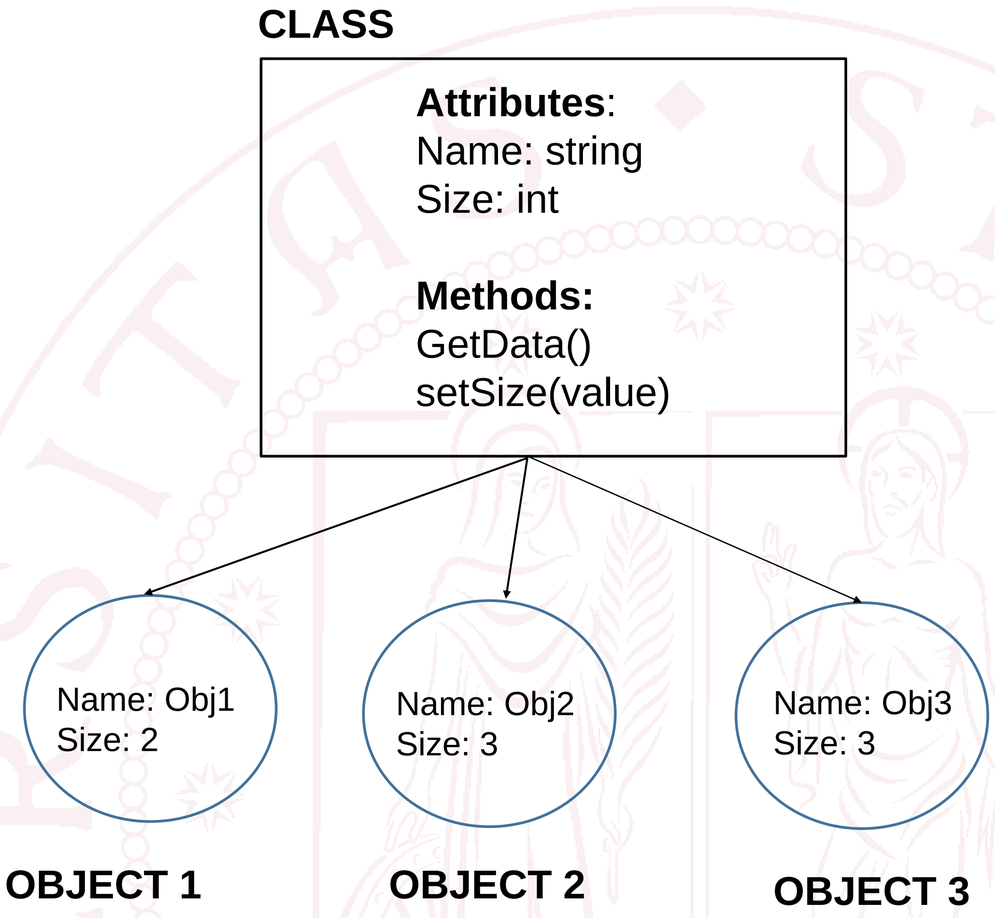
Elements of object-oriented programming?

Class: user-defined data type containing attributes and methods of the associated entity (e.g., monkey)

Objects: instances of the class with specific data

Methods: functions that are defined inside a class that describe the behaviors of an object

Attributes: properties that are defined inside a class that describe the state of an object



Example: Class Student

Let's start by defining the class Student that is characterized by name, surname, ID, degree.

```
class Student:
```

CONSTRUCTOR

```
 def __init__(self, name, surname, id, degree):
    self.name = name
    self.surname = surname
    self.id = id
    self.degree = degree
```

self represents the instance of the class

self is given in input in the constructor to initialize and activate the attributes of the corresponding object

The variables representing the attributes of the entity are called variables of instance

The values of the attributes corresponds to the ones given in input via parameters

Example: Class Student

Let’s define two instances of the class Student:

```
student = Student('Mario', 'Rossi', 10000, 'Computer Science Engineering')
student1 = Student('Francesca', 'Bianchi', 10001, 'Information Engineering')
```

```
print(student)
print(student1)
```

```
<__main__.Student object at 0x7f3673fec190>
<__main__.Student object at 0x7f3673feccd0>
```

The two objects are created and allocated in the **memory**

Example: Class Student

It is different with objects:

```
print(student.name)
print(student.surname)
print(student.id)
print(student.degree)
print("-----------")
print(student1.name)
print(student1.surname)
print(student1.id)
print(student1.degree)
```

```
Mario
Rossi
10000
Computer Science Engineering
-----------
Francesca
Bianchi
10001
Information Engineering
```

We must access to each attribute

To access to one attribute, just using:

Object_name.attribute_name

With the same syntax, you can change the value of attributes

```
student1.degree = 'Electronic Engineering'
print(student1.name + " " + student1.surname + " " + str(student1.id) + " " + student1.degree)
```

```
Francesca Bianchi 10001 Electronic Engineering
```

Example: Class Student

Another solution is to add a method in the class to print the attributes:

```
class Student:

 def __init__(self, name, surname, id, degree):
   self.name = name
   self.surname = surname
   self.id = id
   self.degree = degree

def print_info(self):
  print(self.name + " " + self.surname + " " + str(self.id) + " " + self.degree)
```

```
print_info(student1)
```

```
Francesca Bianchi 10001 Electronic Engineering
```

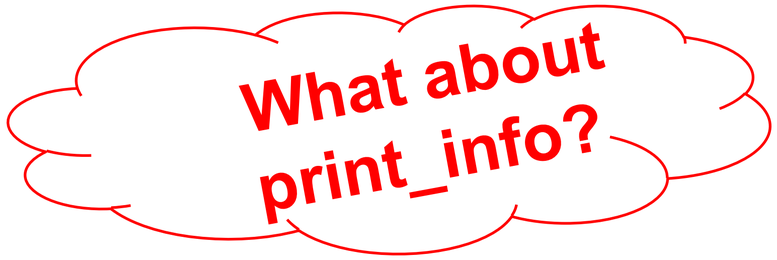
Each method of a class has self (i.e., the instance of the class) as first parameter

This way, the method can be called on each object

Object-oriented programming: Methods

Accessor methods: access the current state of an object without modifying it

Mutator methods: modify the object



ACCESSOR

```
class Student:

 def __init__(self, name, surname, id, degree):
   self.name = name
   self.surname = surname
   self.id = id
   self.degree = degree

def print_info(self):
  print(self.name + " " + self.surname + " " + str(self.id) + " " + self.degree)
```

```
print_info(student1)
```

```
Francesca Bianchi 10001 Electronic Engineering
```

Object-oriented programming: Operators Overloading

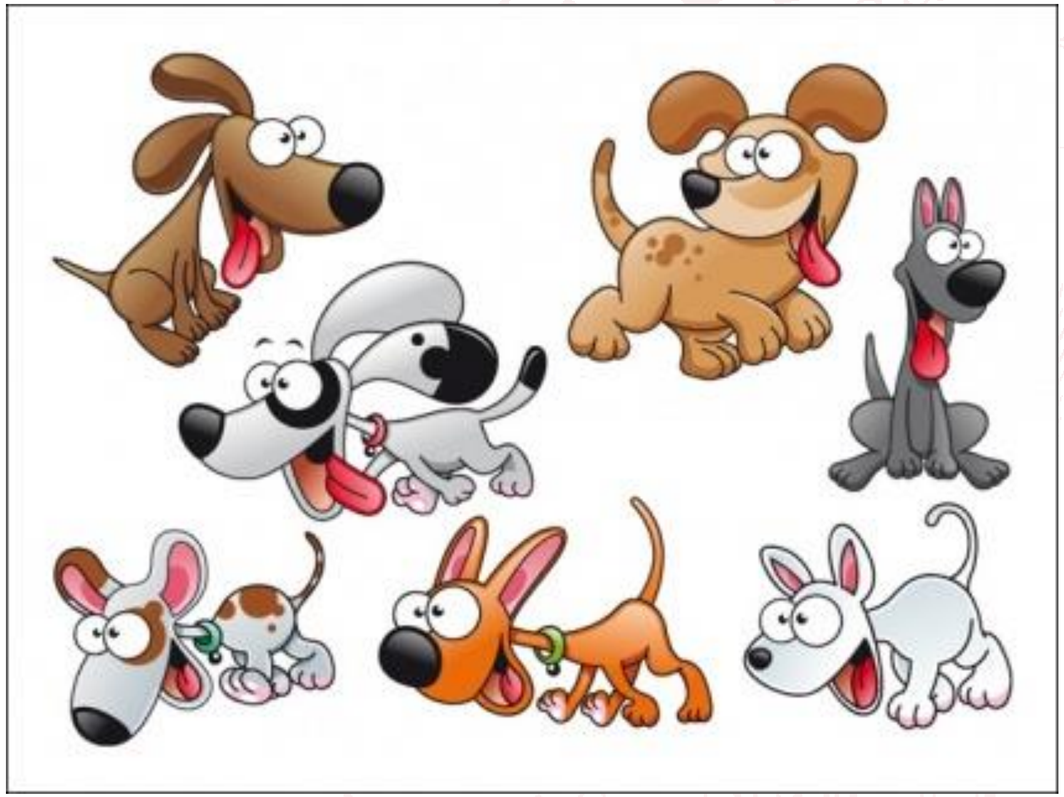
Python allows the operators overloading: namely, if we call a method with the name of an operator, the operator is overwritten, and **the class supports this operator with a different "functionality"**.

For instance, let's consider the class Dog

```
class Dog:
    def __init__(self, name, month, day, year, speakText):
        self.name = name
        self.month = month
        self.day = day
        self.year = year
        self.speakText = speakText
```

Fuffy: 5-15-2004
"BauBau"

Bobby: 10-2-2001
"Auuuuhh"



Ralph: 1-1-2010
"BuffBuff"

Object-oriented programming: Operators Overloading

```
class Dog:
    def __init__(self, name, month, day, year, speakText):
        self.name = name
        self.month = month
        self.day = day
        self.year = year
        self.speakText = speakText

    def speak(self):
        return self.speakText

    def getName(self):
        return self.name

    def birthDate(self):
        return str(self.month) + "/" + str(self.day) + "/" + str(self.year)

    def changeBark(self,bark):
        self.speakText = bark


    def __add__(self, otherDog):
        return Dog("Puppy of " + self.name + " and " + otherDog.name, \
                   self.month, self.day, self.year + 1, \
                   self.speakText + otherDog.speakText)
```

```
boyDog = Dog("Mesa", 5, 15, 2004, "WOOOOF")
girlDog = Dog("Sequoia", 5, 6, 2004, "barkbark")
print(boyDog.speak())
print(girlDog.speak())
print(boyDog.birthDate())
print(girlDog.birthDate())
boyDog.changeBark("arf arf")
print(boyDog.speak())
puppy = boyDog + girlDog
print(puppy.speak())
print(puppy.getName())
print(puppy.birthDate())
```

```
WOOOOF
barkbark
5/15/2004
5/6/2004
arf arf
arf arfbarkbark
Puppy of Mesa and Sequoia
5/15/2005
```

Let's use the overloading of the operator + to define a puppy as boyDog + girlDog

Object-oriented programming: Operators Overloading

__add__(self,y)	x+y	The addition of two objects. The type of x determines which add operator is called.
__contains__(self,y)	y in x	When x is a collection you can test to see if y is in it.
__eq__(self,y)	x == y	Returns True or False depending on the values of x and y .
__ge__(self,y)	x >= y	Returns True or False depending on the values of x and y .
__getitem__(self,y)	x[y]	Returns the item at the y th position in x.
__gt__(self,y)	x>y	Returns True or False depending on the values of x and y .
__hash__(self)	hash(x)	Returns an integral value for x .
__int__(self)	int(x)	Returns an integer representation of x .
__del__(self)	del x	Executed when instance is deleted from memory
__iter__(self)	for v in x	Returns an iterator object for the sequence x .
__le__(self,y)	x <= y	Returns True or False depending on the values of x and y .
__len__(self)	len(x)	Returns the size of x where x has some length attribute.
__lt__(self,y)	x<y	Returns True or False depending on the values of x and y .
__mod__(self,y)	x%y	Returns the value of x modulo y . This is the remainder of x/y .
__mul__(self,y)	x*y	Returns the product of x and y .
__ne__(self,y)	x != y	Returns True or False depending on the values of x and y .
__neg__(self)	-x	Returns the unary negation of x .
__repr__(self)	repr(x)	Returns a string version of x suitable to be evaluated by the eval function.
__setitem__(self,i,y)	x[i] = y	Sets the item at the i th position in x to y .
__str__(self)	str(x)	Return a string representation of x suitable for user-level interaction.
__sub__(self,y)	x-y	The difference of two objects.

Object-oriented programming: Parameters passing

Before explaining the parameters passing, let's analyze what happens in Python during variable assignments:

```
a = 3
b = a
b -=1
print(a)
print(b)
```

```
3
2
```

```
l1 = [1,2,3,4]
l2 = l1
l2.append(5)
print(l1)
print(l2)
```

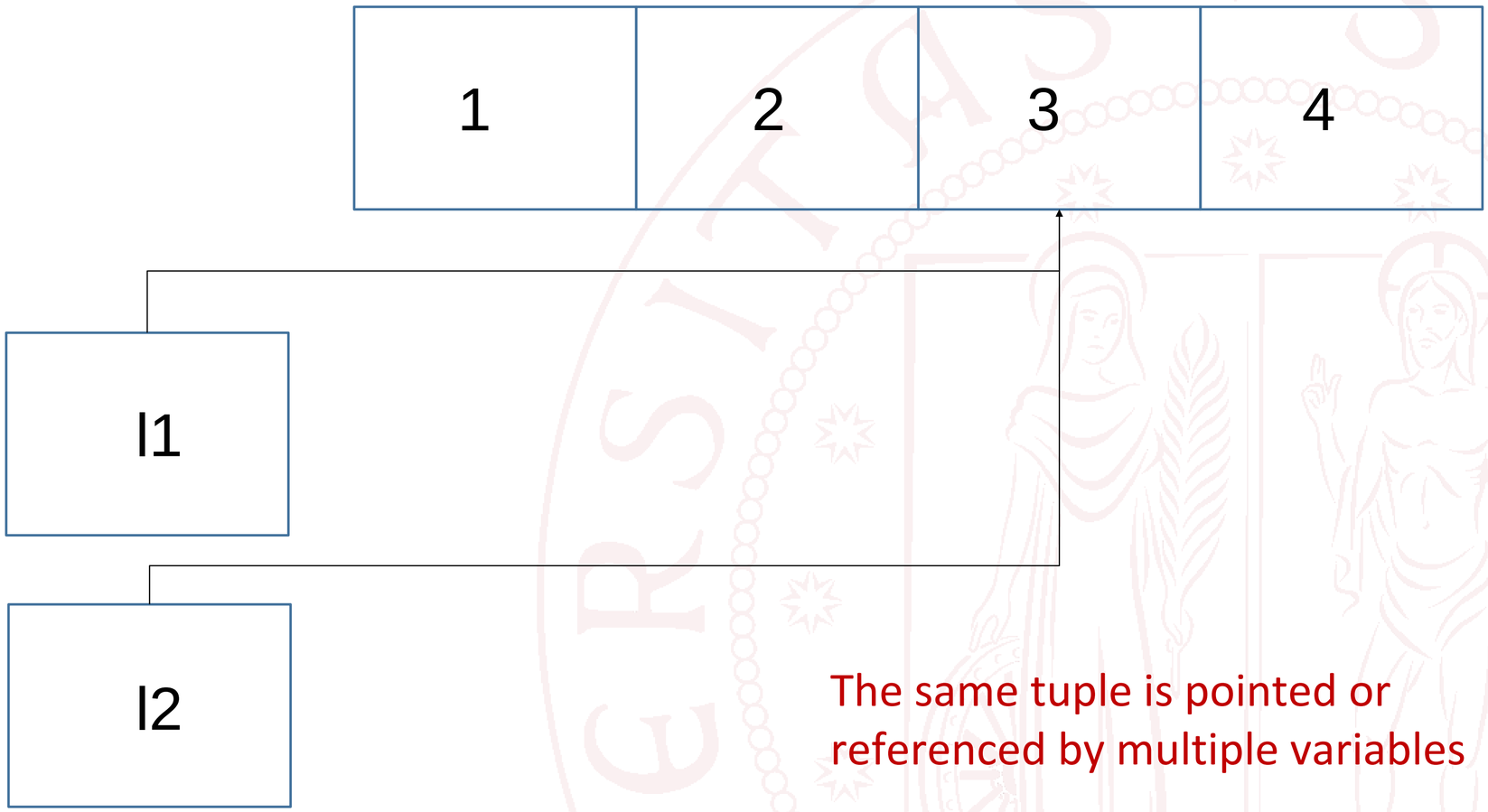
```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

Object-oriented programming: Parameters passing

Let's discussing the second example:

```
l1 = [1,2,3,4]
l2 = l1
l2.append(5)
print(l1)
print(l2)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```



The same tuple is pointed or referenced by multiple variables

Object-oriented programming: Parameters passing

In Python:

- The assignment of a variable only means that the variable points to an object, and does not mean that the object is copied to the variable;

- An object can be pointed to by multiple variables.

- Changes to **mutable objects** (lists, dictionaries, sets, etc.) affect all variables that point to that object.

- For **immutable objects** (strings, ints, tuples, etc.), the value of all variables that point to the object is always the same and does not change.
But when the value of an **immutable object is updated by some operation** (e.g., +=, etc.), **a new object is returned**.

Object-oriented programming: Parameters passing

Let’s consider this example now with istances of the class Jacket:

```
class Jacket():
    def __init__(self):
        self.colour  = "no_colour"
        self.buttons = 1
        self.style   = "unisex"
```

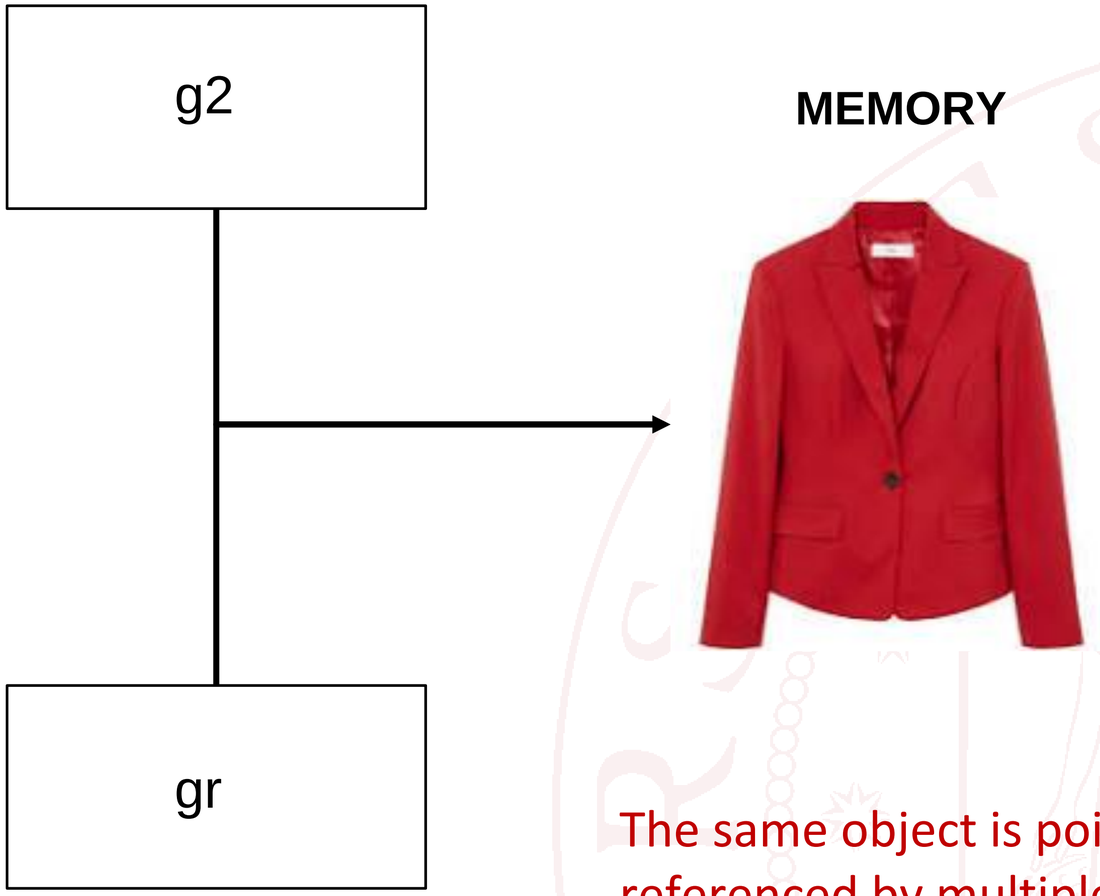
```
g1 = Jacket()
g1.colour = "grey"
g1.buttons = 1
g1.style = “woman"
```

```
g1 = Jacket()
g1.colour = “red"
g1.buttons = 1
g1.style = “woman"
```

Object-oriented programming: Parameters passing

Let’s consider this example now with istances of the class Jacket:

```
g1 = Jacket()
g1.colour = "grey"
g1.buttons = 1
g1.style = "woman"

g2 = Jacket()
g2.colour = "red"
g2.buttons = 1
g2.style = "woman"

gr = g2
gr.colour = "RED"
print(g2.colour)
```



The same object is pointed or referenced by multiple variables

NB: In general, all classes defined by us with **the class construct** are **mutable**

Object-oriented programming: Parameters passing

Python's argument passing is passed by assignment or pass by object reference.

All data types in Python are objects.

So, when passing parameters, just let the **new variable and the original variable point to the same object**, and there is no such thing as passing by value or passing by reference.

Let's consider the following examples:

```
def my_func1(b):
  b = 2

a = 1
my_func1(a)
print(a)
```

```
1
```

IMMUTABLE:
the value is not modified

```
def my_func2(l2):
  l2.append(4)

l1 = [1, 2, 3]
my_func2(l1)
print(l1)
```

```
[1, 2, 3, 4]
```

MUTABLE:
the value is modified

- If an object is **mutable**, when it changes, **all variables** that point to this object change.

- If the object is **immutable**, a simple assignment **can only change** the value of **one** of the variables, leaving the rest of the variables unaffected.

Object-oriented programming: Parameters passing

How to modify the immutable parameters?

```
def my_func1():
  global a
  a = 2

a = 1
my_func1()
print(a)
```

```
2
```

Defining the variables as **global**

```
def my_func1(b):
  return b+1

a = 1
a = my_func1(a)
print(a)
```

```
2
```

Using **return**, to create a new object

Object-oriented programming: Cloning in Python

Two kinds of cloning in Python:

- **Deep cloning**: A deep copy (or deep clone) occurs when the object is copied, and all items in the object are copied as new items.
- **Shallow cloning:** A shallow copy (or shallow clone) occurs when the object is copied, but items in the object are shared with the clone.

The **module copy** supports both deep and shallow cloning

When working with a **shallow clone** of an object that contains mutable elements, the programmer must be aware that the **elements could change** values **without any calls to a method** on the object.

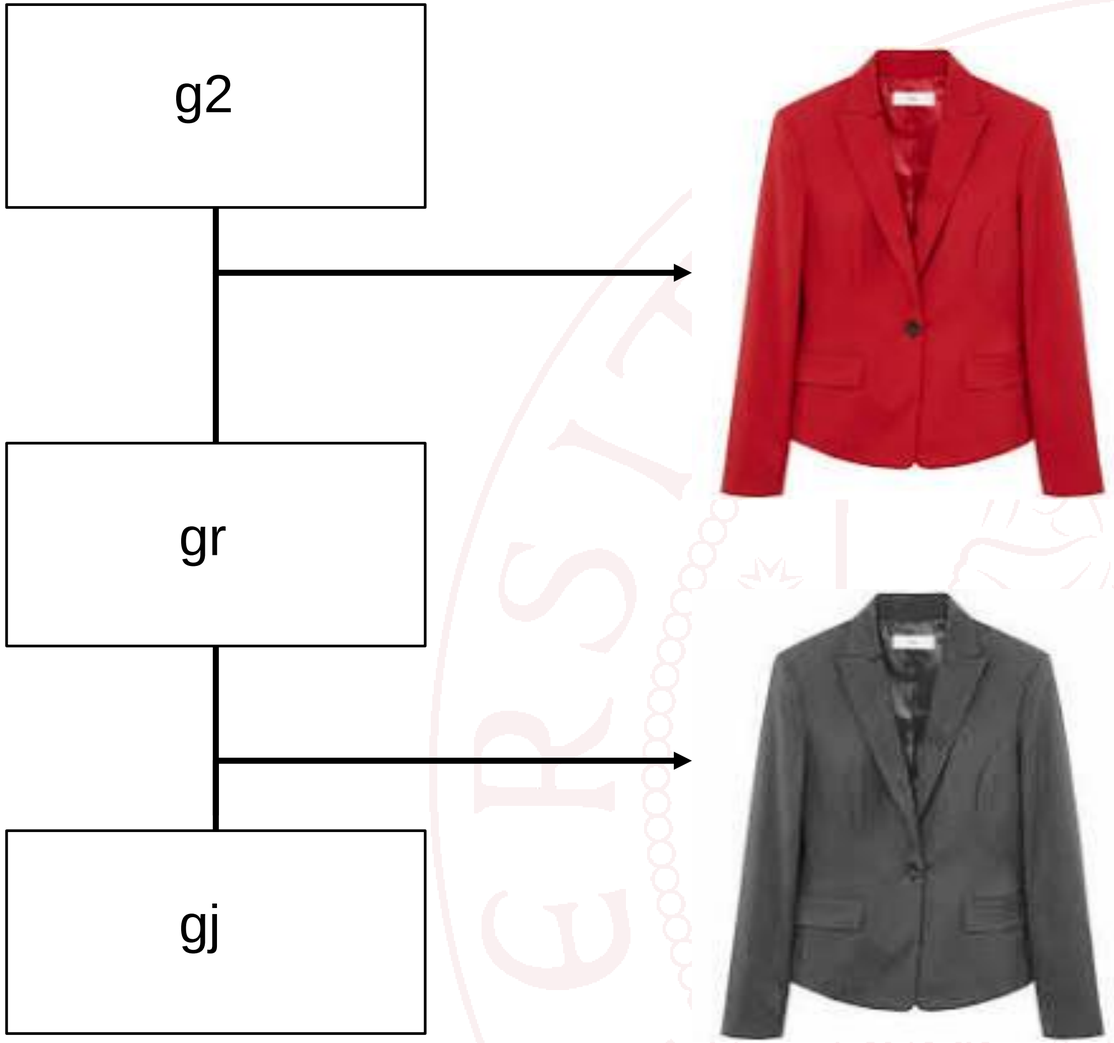
NB: The difference between shallow and deep copy **is only relevant for compound objects** (objects that contain other objects, such as lists or class instances)

Object-oriented programming: Cloning in Python

NB: The difference between shallow and deep copy **is only relevant for compound objects** (objects that contain other objects, such as lists or class instances)

```
import copy
gr = copy.copy(g2)
gr.colour = "BLACK"
print(g2.colour)
print(gr.colour)

gj = copy.deepcopy(g2)
gj.colour = "BLACK"
print(g2.colour)
print(gj.colour)
```

```
RED
BLACK
RED
BLACK
```



Object-oriented programming: Cloning in Python

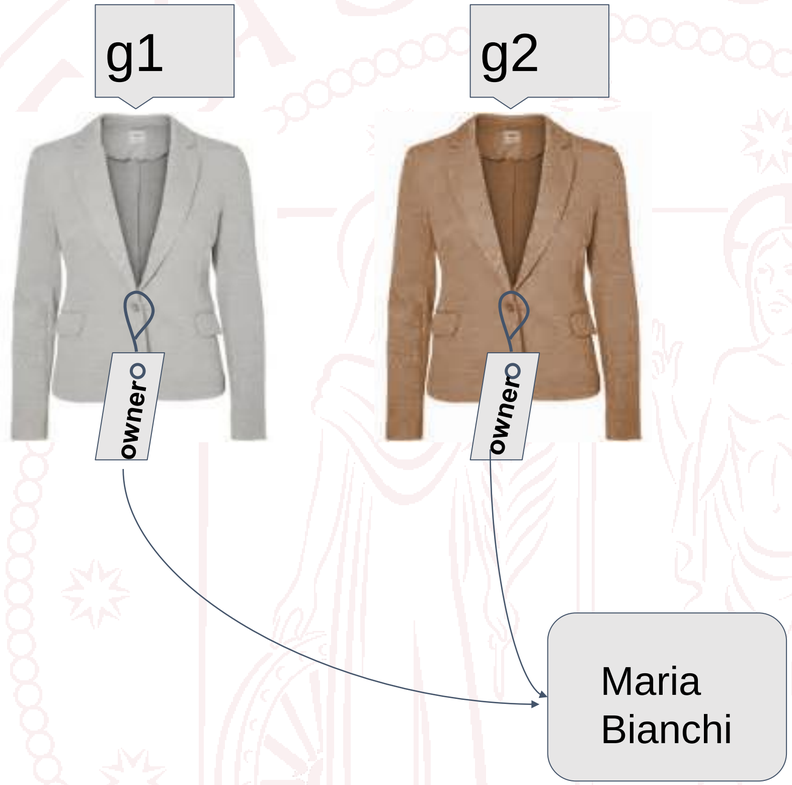
NB: The difference between shallow and deep copy **is only relevant for compound objects** (objects that contain other objects, such as lists or class instances)

```
class Owner():
  def __init__(self):
    self.name = ""

class Jacket():
  def __init__(self):
    self.colour = "no_colour"
    self.buttons = 1
    self.style = "unisex"
```

SHALLOW CLONING

```
import copy

g1 = Jacket()
g1.colour = "grey"
g1.buttons = 1
g1.style = "woman"

p1 = Owner()
p1.name = "Maria Rossi"
g1.owner = p1
print(g1.owner.name)

g2 = copy.copy(g1)
g2.colour = "brown"

g2.owner.name = "Maria Bianchi"

print(g1.owner.name)
print(g2.owner.name)
```

```
Maria Rossi
Maria Bianchi
Maria Bianchi
```



Object-oriented programming: Cloning in Python

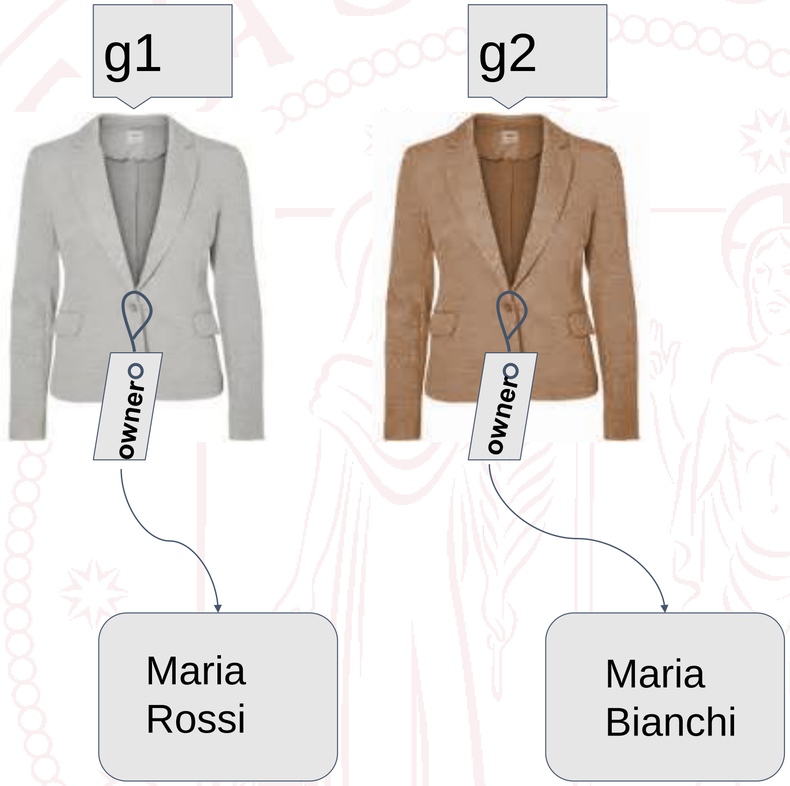
NB: The difference between shallow and deep copy **is only relevant for compound objects** (objects that contain other objects, such as lists or class instances)

```
class Owner():
  def __init__(self):
    self.name = ""

class Jacket():
  def __init__(self):
    self.colour = "no_colour"
    self.buttons = 1
    self.style = "unisex"
```

DEEP CLONING

```
import copy

g1 = Jacket()
g1.colour = "grey"
g1.buttons = 1
g1.style = "woman"

p1 = Owner()
p1.name = "Maria Rossi"
g1.owner = p1
print(g1.owner.name)

g2 = copy.deepcopy(g1)
g2.colour = "brown"

g2.owner.name = "Maria Bianchi"

print(g1.owner.name)
print(g2.owner.name)
```

```
Maria Rossi
Maria Rossi
Maria Bianchi
```



Object-oriented programming: Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

```
print(len('Hello guys!'))
print(len([2,4,5,[5,6]]))
```

```
11
4
```

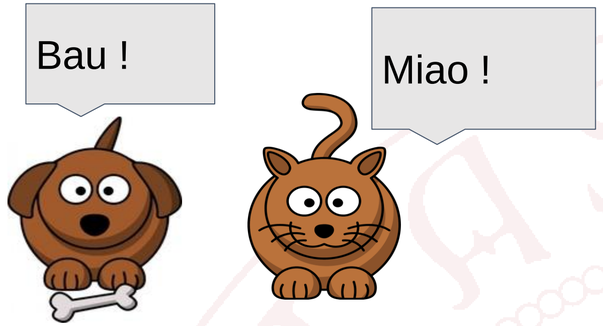
Two examples that we frequently use:
print and len

Object-oriented programming: Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

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

  def say_hello(self):
    print("%s say Bau!" % self.name)
```

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

  def say_hello(self):
    print("%s say Miao!" % self.name)
```




```
my_pets = [Dog('Pluto'), Cat('Fuffy')]
for pet in my_pets:
  pet.say_hello()
```

```
Pluto say Bau!
Fuffy say Miao!
```

We call methods without being concerned about which class type each object is.

Object-oriented programming: Inheritance

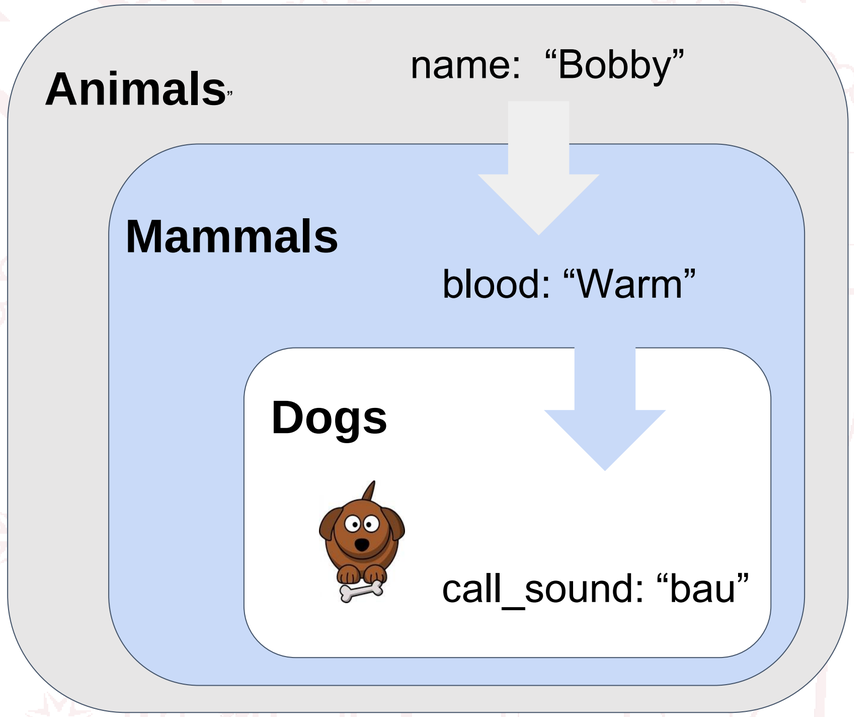
Inheritance allows us to define a class that **inherits all the methods and properties** from another class.

Parent class is the class **being inherited from**, also called base class.

Child class is the class that **inherits from another class**, also called derived class

Multiple advantages:

- Inheritance represents real-world **hierarchical relationships** well
- Inheritance provides **reusability of a code**.
  We don't have to write the same code again and again. It also allows us to add more functionality to a class without changing it.
- Inheritance is **transitive**, which means that if class B inherits from another class A, all subclasses of B will automatically inherit from class A.



Object-oriented programming: Inheritance

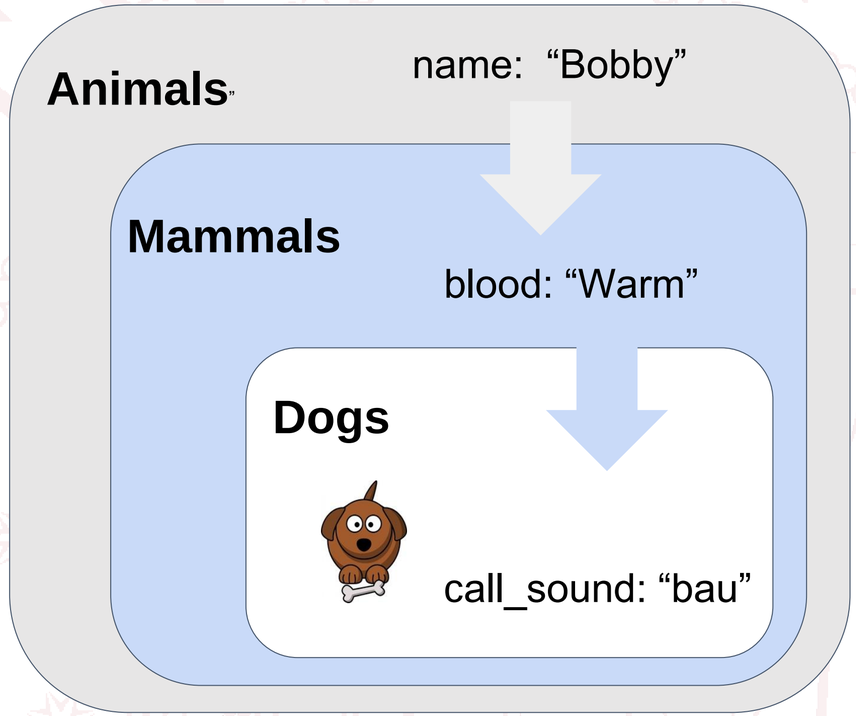
To define a derived class (child class) it is sufficient to **pass it as an argument the name** of the base class (parent class).

A derived class will have **all the features originally associated with the base class** available and these can be used to operate on its instances.

```
class Animal:

 def print_who_i_am(self):
   print("I am an animal")

class Mammal(Animal):
  pass

class Dog(Mammal):
  pass
```

```
def main():
  dog = Dog()
  dog.print_who_i_am()

if __name__== "__main__":
  main()
```

```
I am an animal
```



NB: Use the **pass** keyword when you do not want to add any other properties or methods to the class.

Object-oriented programming: Inheritance

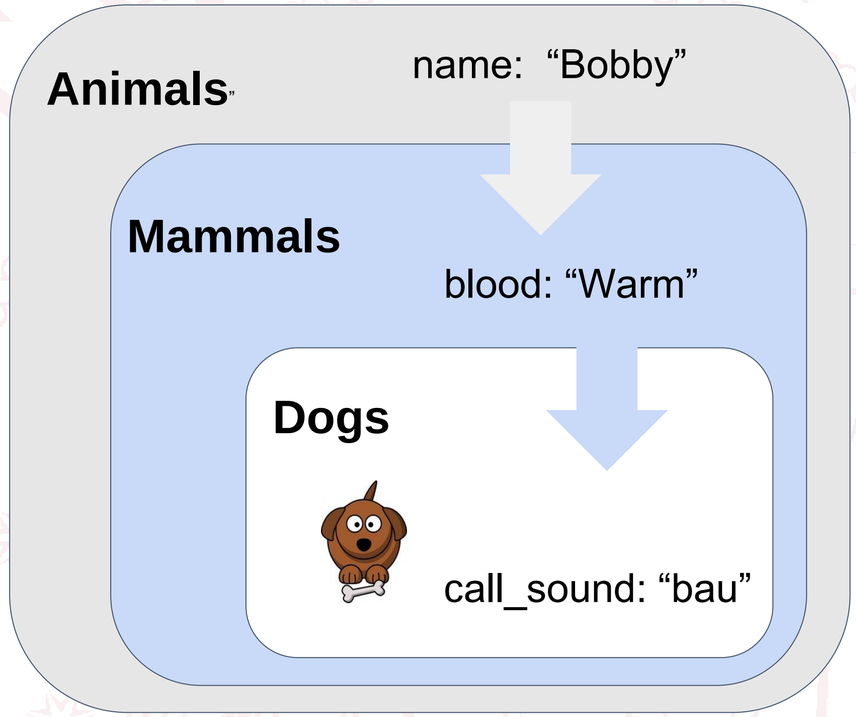
To define a derived class (child class) it is sufficient to **pass it as an argument the name** of the base class (parent class).

A derived class will have **all the features originally associated with the base class** available and these can be used to operate on its instances.

```
class Animal:

 def print_who_i_am(self):
   print("I am an animal")

class Mammal(Animal):

  def print_my_family(self):
    print("I am a mammal")


class Dog(Mammal):

  def print_me(self):
    self.print_who_i_am()
    self.print_my_family()
    print("I am a dog")
```

```
def main():
  dog = Dog()
  dog.print_me()

if __name__== "__main__":
  main()
```

```
I am an animal
I am a mammal
I am a dog
```



NB: Use **self** to call the inherited methods

Object-oriented programming: Inheritance

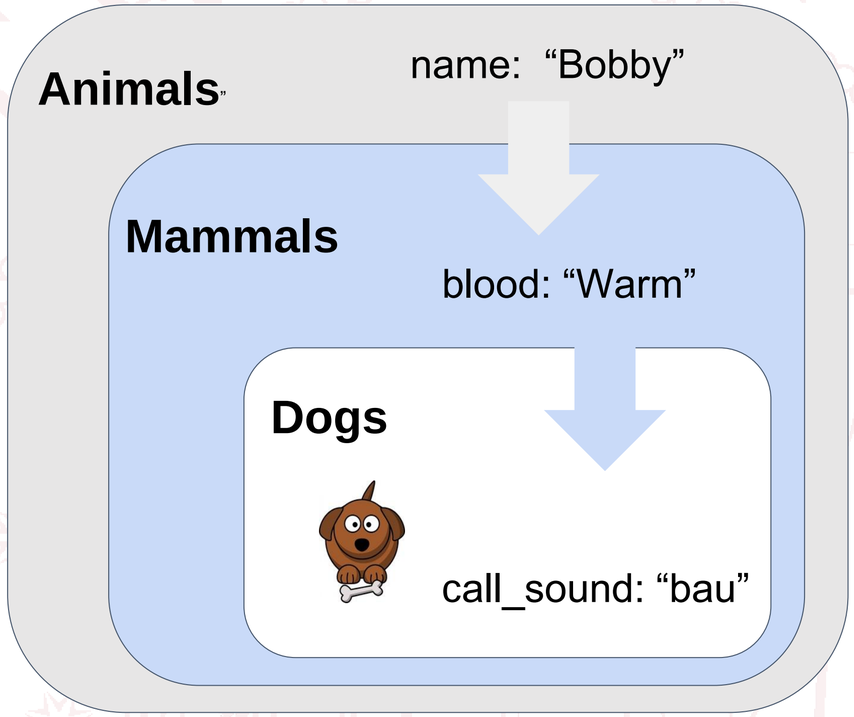
To define a derived class (child class) it is sufficient to **pass it as an argument the name** of the base class (parent class).

A derived class will have **all the features originally associated with the base class** available and these can be used to operate on its instances.

```
class Animal:

 def print_who_i_am(self):
   print("I am an animal")

class Mammal(Animal):

  def print_who_i_am(self):
    print("I am a mammal")


class Dog(Mammal):

  def print_who_i_am(self):
    print("I am a dog")
```

```
def main():
  dog = Dog()
  dog.print_who_i_am()

if __name__== "__main__":
  main()
```

```
I am a dog
```



NB: You can **override** the inherited methods

overloading when the new method and the existing method differ in the number or types of arguments (the method signature changes - same name but different arguments).

overriding when the signature remains unchanged (same name and same arguments) between parent class and child class.

Object-oriented programming: Inheritance

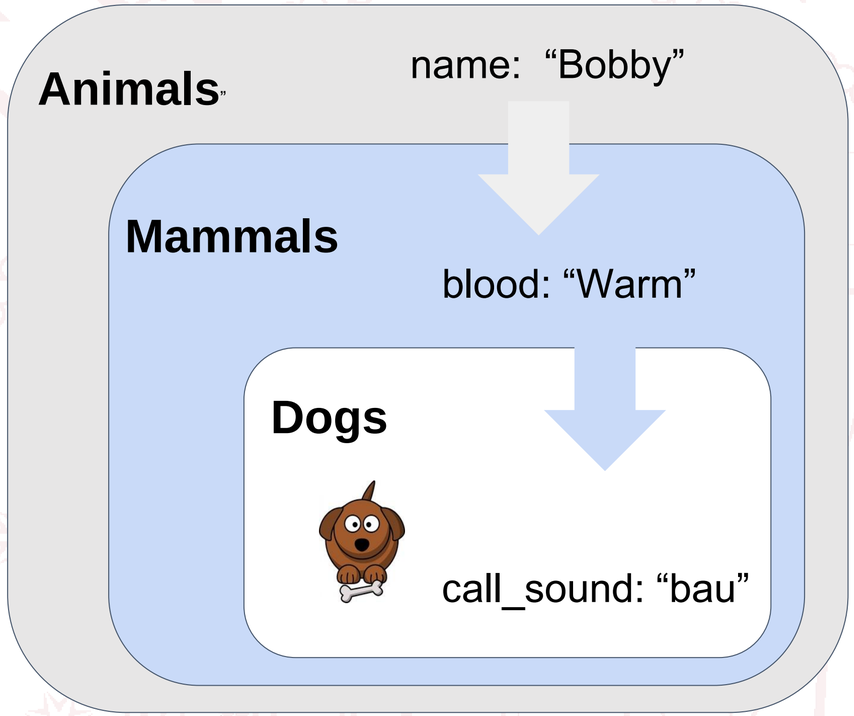
To define a derived class (child class) it is sufficient to **pass it as an argument the name** of the base class (parent class).

A derived class will have **all the features originally associated with the base class** available and these can be used to operate on its instances.

```
class Animal:

  def __init__(self, name = None):
    self.name = name
    self.blood = ""

  def print_me(self):
    if self.name:
      print("I am %s" % self.name)
      print("I have a %s blood" % self.blood)

class Mammal(Animal):

  def __init__(self, name = None):
    Animal.__init__(self, name)
    self.blood = "warm"

class Dog(Mammal):

  def __init__(self, name = None):
    Mammal.__init__(self, name)
    self.call_sound = "bau"

  def print_me(self):
    Mammal.print_me(self)
    print("My call sound is %s" % self.call_sound)
```

```
def main():
  dog = Dog("Bobby")
  dog.print_me()

if __name__== "__main__":
  main()
```

```
I am Bobby
I have a warm blood
My call sound is bau
```



NB: It works also with the **constructor**

Object-oriented programming: Inheritance

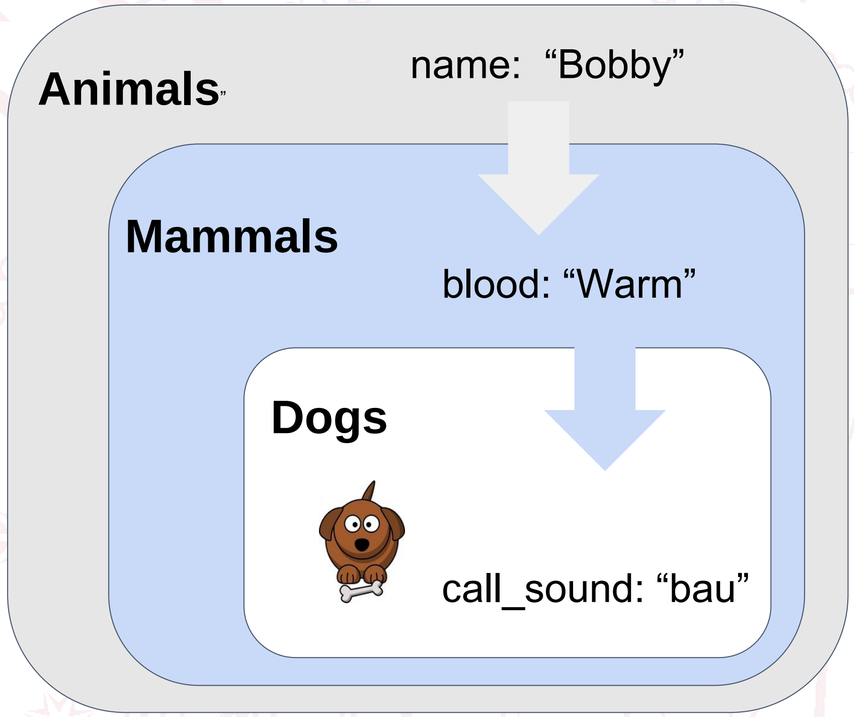
To define a derived class (child class) it is sufficient to **pass it as an argument the name** of the base class (parent class).

A derived class will have **all the features originally associated with the base class** available and these can be used to operate on its instances.

```
class Animal:

  def __init__(self, name = None):
    self.name = name
    self.blood = ""

  def print_me(self):
    if self.name:
      print("I am %s" % self.name)
      print("I have a %s blood" % self.blood)

class Mammal(Animal):

  def __init__(self, name = None):
    Animal.__init__(self, name)
    self.blood = "warm"

class Dog(Mammal):

  def __init__(self, name = None):
    Mammal.__init__(self, name)
    self.call_sound = "bau"

  def print_me(self):
    Mammal.print_me(self)
    print("My call sound is %s" % self.call_sound)
```

```
def main():
  dog = Dog("Bobby")
  dog.print_me()

if __name__== "__main__":
  main()
```

```
I am Bobby
I have a warm blood
My call sound is bau
```



NB: We can call methods of the parent classes with type.method_name(args)

Object-oriented programming: Inheritance

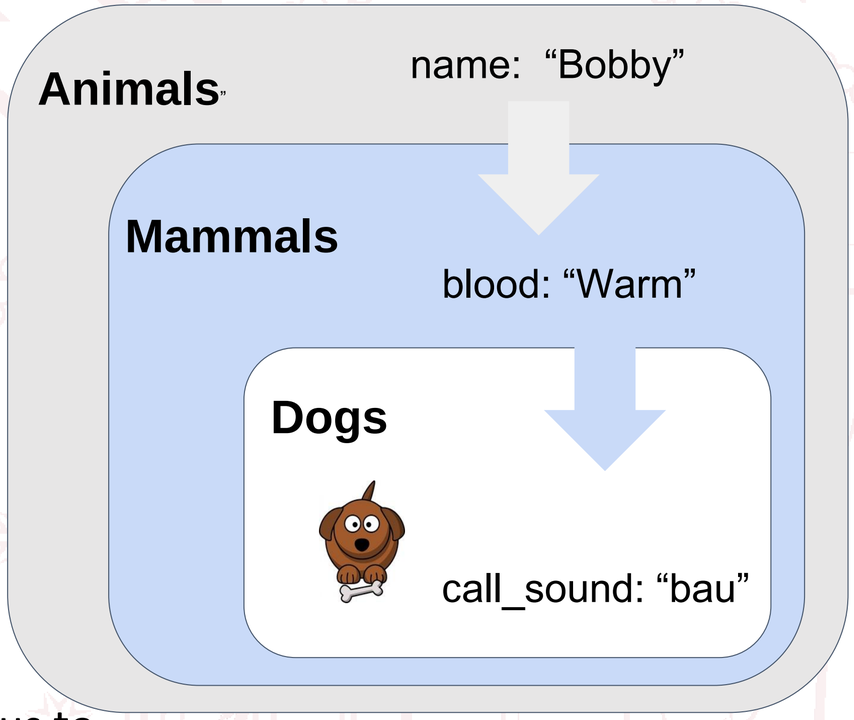
To define a derived class (child class) it is sufficient to **pass it as an argument the name** of the base class (parent class).

A derived class will have **all the features originally associated with the base class** available and these can be used to operate on its instances.

```
class Animal:

  def __init__(self, name = None):
    self.name = name
    self.blood = ""

  def print_me(self):
    if self.name:
      print("I am %s" % self.name)
      print("I have a %s blood" % self.blood)

class Mammal(Animal):

  def __init__(self, name = None):
    super().__init__(name)
    self.blood = "warm"

class Dog(Mammal):

  def __init__(self, name = None):
    super().__init__(name)
    self.call_sound = "bau"

  def print_me(self):
    super().print_me()
    print("My call sound is %s" % self.call_sound)
```

```
def main():
  dog = Dog("Bobby")
  dog.print_me()

if __name__== "__main__":
  main()
```

```
I am Bobby
I have a warm blood
My call sound is bau
```



NB: super() returns a proxy object (temporary object of the superclass) that allows us to access methods of the base class.

Questions