

# 13 – Programación Orientada a Objetos en Python 3



Diego Andrés Alvarez Marín  
Profesor Asociado

Universidad Nacional de Colombia  
Sede Manizales

# Tabla de contenido

- Historia
- Vocabulario
- La clase mínima
- Atributos de la clase y del objeto

Nota: la mayoría de las diapositivas mostradas aquí están basadas en el material de programación orientada a objetos que se encuentra en:

- <https://www.python-course.eu>
- Voy en:
- [https://www.python-course.eu/python3\\_multiple\\_inheritance.php](https://www.python-course.eu/python3_multiple_inheritance.php)

# La programación orientada a objetos (POO)

- Apareció con el lenguaje Simula 67 en 1967 y posteriormente dicho paradigma fue mejorado con el lenguaje Smalltalk en los 1970s.
- El lenguaje C++ y el lenguaje Java popularizaron su uso, especialmente, porque la POO se utiliza frecuentemente para hacer los entornos gráficos de los programas (GUIs).
- Python es un lenguaje primariamente orientado a objetos, ya que en este todas las entidades son objetos.

# Vocabulario

- Un objeto es una realización de una clase. Por ejemplo, en un juego de estrategia, consideremos la clase “guerrero”. En el contexto de POO cada guerrero del juego sería un objeto, cada uno de ellos con atributos (edad, salud, coraza, etc) y métodos (caminar, pelear, comer, etc.)
- Los datos que pertenecen a un objeto se les llama los "atributos (o propiedades) del objeto". En un programa orientado a objetos, estos están ocultos a través de una interfase, y solo se puede acceder a los atributos del objeto a través de funciones especiales, a las cuales se les llama métodos en el contexto de la POO. El poner los datos detrás de la interfase se le llama encapsulación.
- En términos generales, un objeto se define por una clase. Una clase es una descripción formal del diseño de un objeto, es decir, especifica los atributos y métodos que el objeto tiene. A estos objetos también se les llama incorrectamente en español instancias (del inglés "instances") o realizaciones. Evite confundir una clase con un objeto. Jorge y María son objetos de la clase “Persona”. Nombre la clase con la primera letra <sub>5</sub> en mayúscula.

# En Python todas las entidades son objetos

```
>>> a = 42
>>> type(a)
<class 'int'>
>>> b = 2.3
>>> type(b)
<class 'float'>
>>> c = lambda x: x+2
>>> type(c)
<class 'function'>
>>> import math
>>> type(math)
<class 'module'>
>>> L = [1,2,3]
>>> type(L)
<class 'list'>
>>> T = ('x','y',3)
>>> type(T)
<class 'tuple'>
>>> cad = "Hola"
>>> type(cad)
<class 'str'>
```

Guido van Rossum escribió:

*"One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth."* (Blog, The History of Python, February 27, 2009)

```
>>> L1 = [1,2,3]
>>> L2 = ['x','y','z']
>>> L1.append(4)
>>> x = L2.pop()
>>> L1
[1, 2, 3, 4]
>>> L2
['x', 'y']
>>> x
'z'
```

Aquí L1 y L2 son objetos, los cuales son realizaciones de la clase “list”. Ellos tienen métodos asociados (append(), pop()).

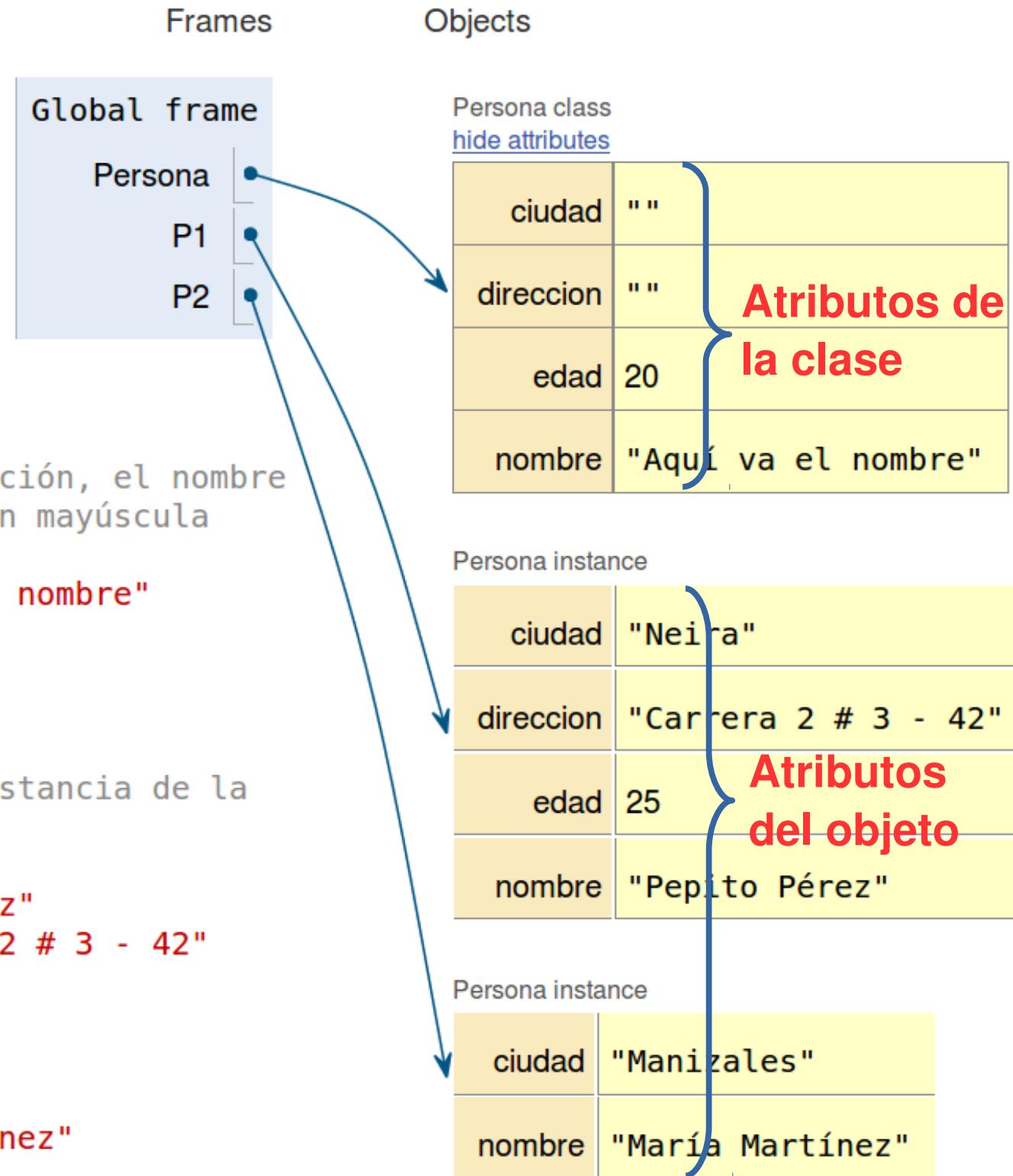
# La clase mínima en Python

```
>>> class Robot:  
        pass  
  
>>> x = Robot()  
>>> y = Robot()  
>>> z = y  
>>> x  
<__main__.Robot object at 0x7f23bdf50da0>  
>>> y  
<__main__.Robot object at 0x7f23bdf50eb8>  
>>> type(x)  
<class '__main__.Robot'>  
>>> type(y)  
<class '__main__.Robot'>  
>>> type(z)  
<class '__main__.Robot'>  
  
>>> x == y  
False  
>>> y == z  
True  
>>> y is z  
True
```

- Robot es la clase
- x, y son objetos
- El operador = copia la referencia.

► Las dos instancias de la clase Robot son diferentes.

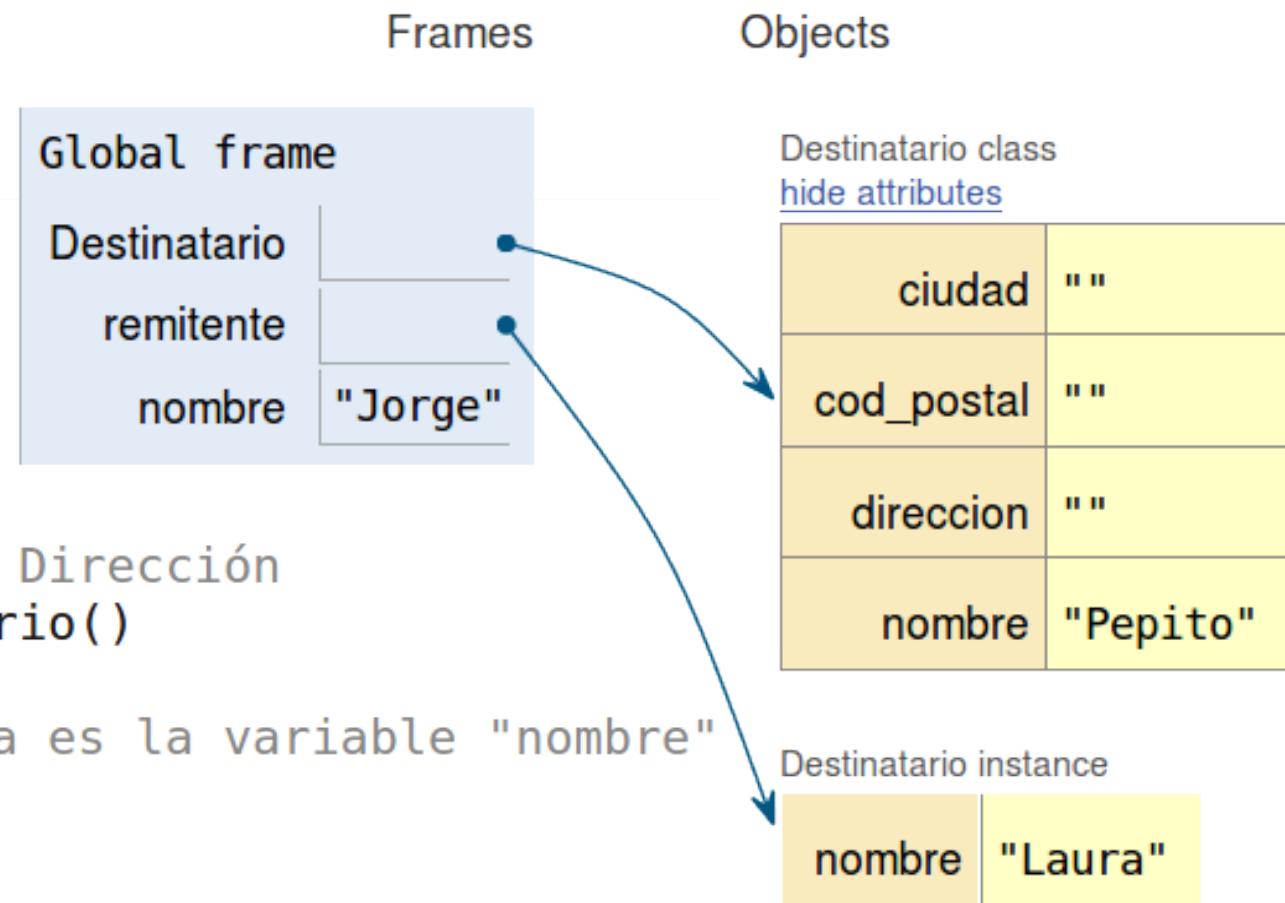
# Clases de solo atributos (similar a las estructuras en C/C++)

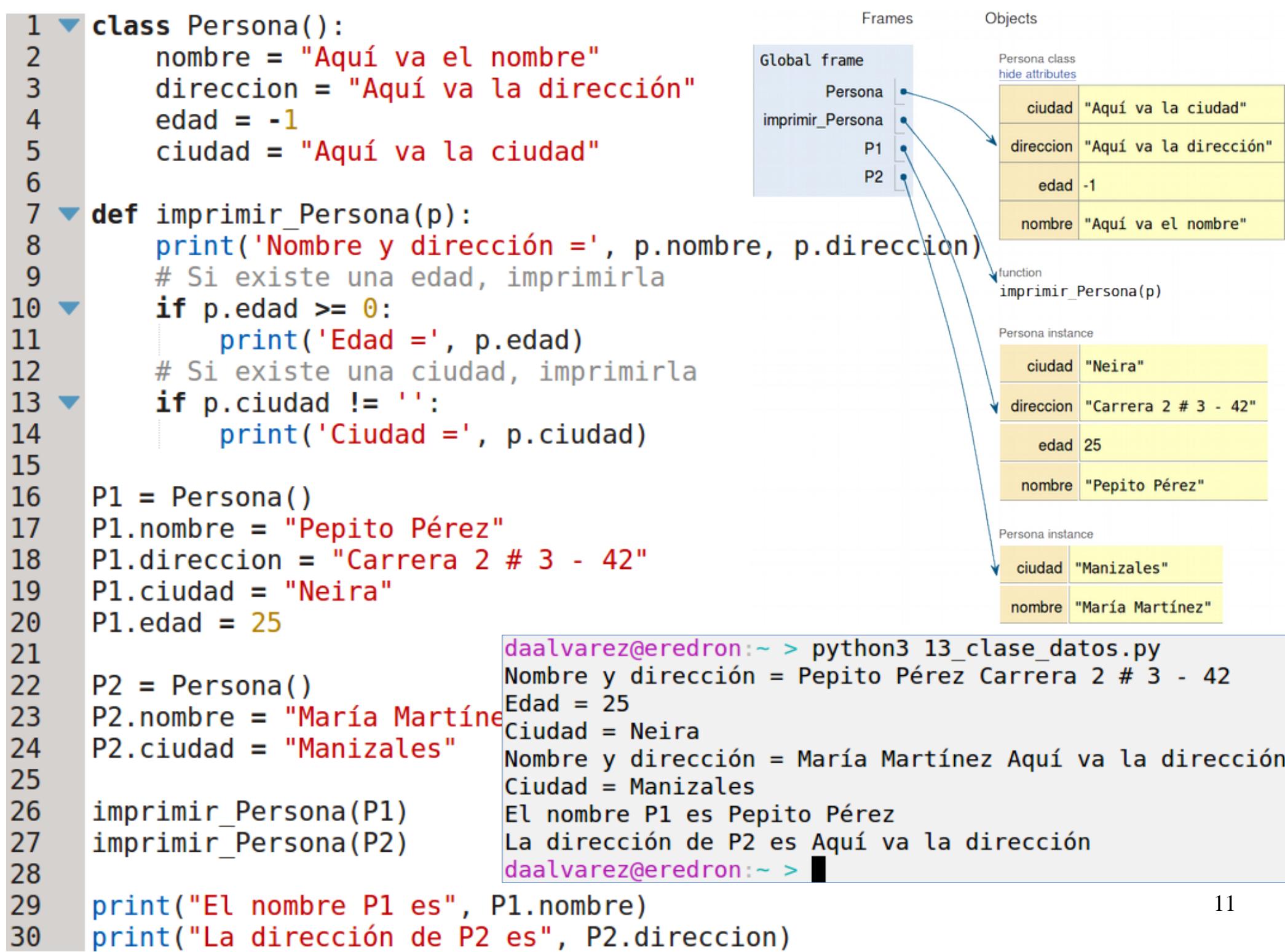


```
1 # Observe que por convención, el nombre
2 # de la clase empieza con mayúscula
3 ▼ class Persona():
4     nombre = "Aquí va el nombre"
5     direccion = ""
6     edad = 20
7     ciudad = ""
8
9     # Se crea un objeto o instancia de la
10    # clase Persona()
11    P1 = Persona()
12    P1.nombre = "Pepito Pérez"
13    P1.direccion = "Carrera 2 # 3 - 42"
14    P1.ciudad = "Neira"
15    P1.edad = 25
16
17    P2 = Persona()
18    P2.nombre = "María Martínez"
19    P2.ciudad = "Manizales"
```

Un problema muy común cuando trabajamos con clases, es no especificar con qué realización de la clase queremos hacerlo. Si solo se ha creado una dirección, es comprensible asumir que el computador sabrá cómo usar la dirección de la que estás hablando. Sin embargo, esto no siempre es así. Observa el siguiente ejemplo:

```
1 class Destinatario():
2     nombre = ""
3     direccion = ""
4     ciudad = ""
5     cod_postal = ""
6
7 # Crea un objeto tipo Dirección
8 remitente = Destinatario()
9
10 # Cuidado! esto crea a es la variable "nombre"
11 nombre = "Jorge"
12
13 # Esto tampoco
14 Destinatario.nombre = "Pepito"
15
16 # Esto si está bien:
17 remitente.nombre = "Laura"
```





# Métodos

Un método es una función cuyo código está definido dentro una clase.

Un método difiere de una función en dos aspectos:

- El método pertenece a una clase.
- El primer parámetro en la definición de un método es una referencia al objeto que llama el método. Este se reconoce con el nombre **self**.

```

1 class Perro():
2     def __init__(self): # el constructor
3         self.edad = 0
4         self.nombre = ""
5         self.peso = 0
6         self.color = 'Negro'
7
8     def ladrar(self, num=1):
9         print(num * "Guau!! ")
10
11    def imprimir(self):
12        print('Nombre =', self.nombre)
13        print('Edad =', self.edad, 'años')
14        print('Peso =', self.peso, 'kg')
15        print('Color =', self.color)
16
17 p = Perro()
18 p.edad = 10 # años
19 p.nombre = 'Guardián'
20 p.peso = 50 # kg
21 p.raza = 'Criollo'
22
23 p.ladrar(3)
24 p.imprimir()

```

Line: 26 of 26 Col: 1 LINE INS

daalvarez@eredron:~ > python3 13\_clases\_metodos.py

Guau!! Guau!! Guau!!

Nombre = Guardián

Edad = 10 años

Peso = 50 kg

Color = Negro

daalvarez@eredron:~ > □

# Agregando métodos a una clase

Global frame

Perro  
p

Perro class  
hide attributes

__init__	function __init__(self)
imprimir	function imprimir(self)
ladrar	function ladrar(self, num)

Perro instance

color	"Negro"
edad	10
nombre	"Guardián"
peso	50
raza	"Criollo"

Esto es equivalente a decir **Perro.imprimir(p)** Observe que aquí **self** se refiere a p. La notación **p.imprimir()** es preferible ya que es la estándar en POO.

# Tres formas diferentes de llamar a un método

```
>>> p
<__main__.Perro object at 0x7ff61ab12cc0>
>>> p.imprimir()
Nombre =
Edad = 0 años
Peso = 0 kg
Color = Negro
>>> Perro.imprimir(p)
Nombre =
Edad = 0 años
Peso = 0 kg
Color = Negro
>>> type(p).imprimir(p)
Nombre =
Edad = 0 años
Peso = 0 kg
Color = Negro
```

# Agregando métodos a una clase

Cuando cree métodos para las clases, tenga en cuenta las siguientes observaciones:

- Primero se deben listar los atributos, luego los métodos.
- El primer parámetro de un método de una clase debe ser `self`. Este parámetro se requiere, incluso si la función no lo usa. `self` hace referencia al objeto que llama el método.
- Como tal llamar el primer parámetro `self` es una convención (`self` no es una palabra reservada); podría usarse otro nombre, sin embargo, es tan extendido su uso, que se recomienda utilizar dicha palabra.
- `self` es como decir el pronombre “mi” (en lenguaje C++ es equivalente a la palabra reservada `this`), ya que dentro de una clase estoy hablando de mi nombre, mi dirección, mi peso, y fuera de la clase estoy hablando del nombre, dirección y peso de ese objeto.
- Las definiciones de los métodos deben identarse.

# El método constructor

En POO, un constructor es un método de la clase cuya misión es inicializar un objeto de una clase. En el constructor se asignan los valores iniciales de los atributos del nuevo objeto, y en ciertos casos prepara el sistema para la creación del nuevo objeto. Este método se invoca automáticamente cada vez que una instancia de una clase es creada.

El constructor en Python es el método  
`__init__()`

```
1 ▼ class Perro():
2     def __init__(self,nombre,color_perro='Negro'):
3         self.nombre = nombre
4         self.color = color_perro
5
6     def ladrar(self, num=1):
7         print(num * "Guau!! ")
8
9     def imprimir(self):
10        print('Nombre =', self.nombre)
11        print('Color =', self.color)
12
13 p1 = Perro('Guardián', 'Blanco')
14 p1.ladrar(3)
15 p1.imprimir()
16 print(p1.nombre, p1.color)
17
18 p2 = Perro('Kaiser')
19 p2.imprimir()
20 print(p2.nombre, p2.color)
```

# Constructores con parámetros

Line: 22 of 22 Col: 1    LINE    INS

```
daa@heimdall ~ $ python3 13_constructores_con_par.py
Guau!! Guau!! Guau!!
Nombre = Guardián
Color = Blanco
Guardián Blanco
Nombre = Kaiser
Color = Negro
Kaiser Negro
```

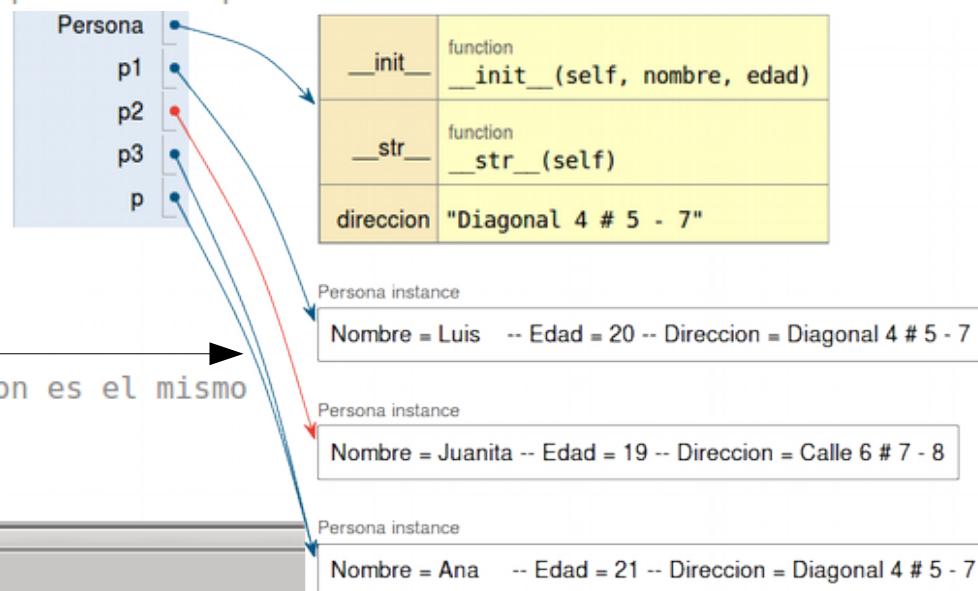
# Atributos del objeto vs. Atributos de la clase

Un **atributo del objeto** es una variable que toma el valor de acuerdo al objeto. Por ejemplo en una habitación llena de gente, cada persona tiene una edad. También se le llama **instance attributes** (en inglés). They are owned by the specific instances of a class. This means for two different instances the instance attributes are usually different.

Los **atributos de la clase** pertenecen a la clase misma; son atributos que son compartidos por todos los objetos de la clase, por lo que es el mismo atributo para cada objeto de la clase. We define class attributes outside of all the methods, usually they are placed at the top, right below the class header. Por ejemplo en un grupo de personas, todas son humanos. Son aquellas variables que se definen dentro de la clase pero fuera de los métodos. También se le llama **class attributes** (en inglés)

# Los atributos del objeto pueden ocultar los atributos de la clase

```
1 class Persona():
2     direccion = 'Carrera 1 # 2 - 3' # atributo de la clase
3     def __init__(self, nombre, edad):
4         self.nombre = nombre # atributo del objeto
5         self.edad = edad     # atributo del objeto
6
7     def __str__(self):
8         return 'Nombre = {0:7} -- Edad = {1:2} -- Direccion = {2}'.format(self.nombre, self.edad, self.direccion)
9
10 p1 = Persona('Luis', 20); p2 = Persona('Juanita', 19); p3 = Persona('Ana', 21)
11
12
13 # se cambia el atributo de la clase únicamente para p2 de modo que ahora
14 # p2.direccion es un atributo del objeto
15 p2.direccion = 'Calle 6 # 7 - 8'
16 for p in [p1, p2, p3]: print(p)
17 print(70*'-')
18
19 # se cambió el atributo de la clase
20 Persona.direccion = 'Diagonal 4 # 5 - 7'
21 for p in [p1, p2, p3]: print(p)
22 print(70*'-')
23
24 # se borra el atributo del objeto y ahora p2.direccion es el mismo
25 # atributo de la clase
26 del p2.direccion
27 for p in [p1, p2, p3]: print(p)
```



Line: 29 of 29 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 13_atributos_clase_vs_atributos_objeto.py
Nombre = Luis    -- Edad = 20 -- Direccion = Carrera 1 # 2 - 3
Nombre = Juanita -- Edad = 19 -- Direccion = Calle 6 # 7 - 8
Nombre = Ana     -- Edad = 21 -- Direccion = Carrera 1 # 2 - 3
-----
Nombre = Luis    -- Edad = 20 -- Direccion = Diagonal 4 # 5 - 7
Nombre = Juanita -- Edad = 19 -- Direccion = Calle 6 # 7 - 8
Nombre = Ana     -- Edad = 21 -- Direccion = Diagonal 4 # 5 - 7
-----
Nombre = Luis    -- Edad = 20 -- Direccion = Diagonal 4 # 5 - 7
Nombre = Juanita -- Edad = 19 -- Direccion = Diagonal 4 # 5 - 7
Nombre = Ana     -- Edad = 21 -- Direccion = Diagonal 4 # 5 - 7
```

```

1 # Example of an instance variable
2 class ClassA():
3     def __init__(self):
4         self.y = 3
5
6 # Example of a static variable
7 class ClassB():
8     x = 7
9
10 # Create class instances
11 a = ClassA()
12 b = ClassB()
13
14 # Two ways to print the STATIC variable.          CAMBIAR TEXTO!!!
15 print(b.x)
16 print(ClassB.x) # This is the proper way to do it.
17
18 # One way to print an INSTANCE variable.
19 # The second generates an error, because we don't know what instance
20 # to reference.
21 print(a.y)      # This is the proper way to do it
22 print(ClassA.y) # This way generates an error

```

- En ClassA y es un atributo del objeto.
- En ClassB x es un atributo de la clase.

Line: 22 of 24 Col: 1    LINE    INS                                  13\_variable\_instancia\_variable\_estatica.py    UT

daalvarez@eredron:~ > python3 13\_variable\_instancia\_variable\_estatica.py

7  
7  
3

Traceback (most recent call last):  
 File "13\_variable\_instancia\_variable\_estatica.py", line 22, in <module>  
 print(ClassA.y) # This way generates an error  
AttributeError: type object 'ClassA' has no attribute 'y'

daalvarez@eredron:~ > █

```

1 # Class with a static variable
2 class ClassB():
3     x = 7
4 # Create a class instance
5 b = ClassB()
6
7 print(b.x)      # This prints 7
8 print(ClassB.x) # This also prints 7
9

```

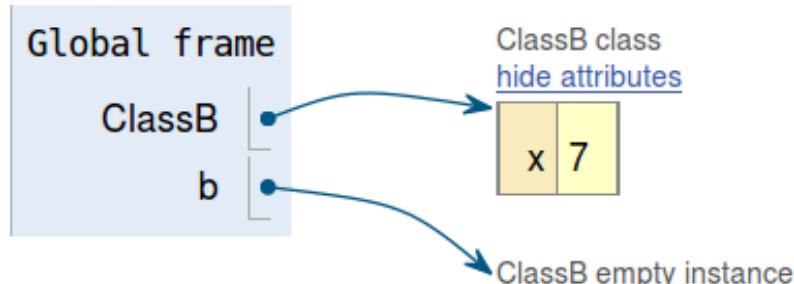
```

10 # Set x to a new value using the class name
11 ClassB.x = 8
12
13 print(b.x)      # This also prints 8
14 print(ClassB.x) # This prints 8
15
16 # Set x to a new value using the instance.
17 # Wait! Actually, it doesn't set x to a new
18 # value! It creates a brand new variable, x.
19 # This x is an instance variable. The static
20 # variable is also called x. But they are
21 # two different variables. This is super-
22 # confusing and is bad practice.
23 b.x = 9
24
25 print(b.x)      # This prints 9
26 print(ClassB.x) # This prints 8. NOT 9!!!

```

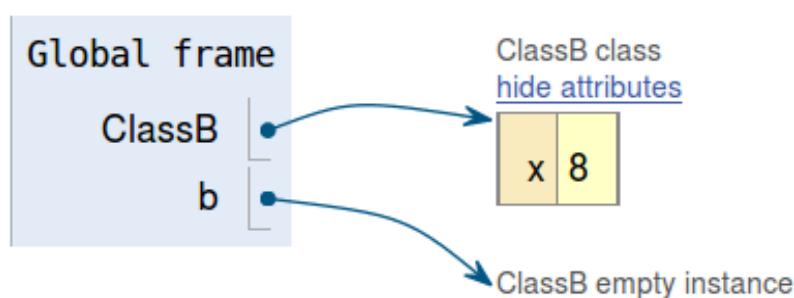
Frames

Objects



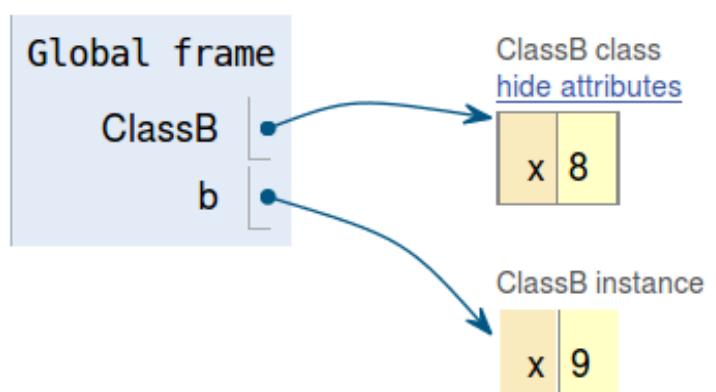
Frames

Objects



Frames

Objects



Line: 28 of 28 Col: 1 LINE INS  
daalvarez@eredron:~ > python3 13\_variables\_instancia\_

7  
7 Los atributos del objeto pueden  
8 ocultar los atributos de la clase  
9  
8

# El atributo `__dict__`

Python's class attributes and object attributes are stored in separate dictionaries, as we can see here:

```
>>> x.__dict__
{'a': "This creates a new instance attribute for x!"}
>>> y.__dict__
{}
>>> A.__dict__
dict_proxy({'a': "This is changing the class attribute 'a'!", '__dict__': <attribute '__dict__' of 'A' objects>, '__module__': '__main__', '__weakref__': <attribute '__weakref__' of 'A' objects>, '__doc__': None})
>>> x.__class__.__dict__
dict_proxy({'a': "This is changing the class attribute 'a'!", '__dict__': <attribute '__dict__' of 'A' objects>, '__module__': '__main__', '__weakref__': <attribute '__weakref__' of 'A' objects>, '__doc__': None})
>>>
```

Ver :

[http://www.python-course.eu/python3\\_class\\_and\\_instance\\_attributes.php](http://www.python-course.eu/python3_class_and_instance_attributes.php)

# El atributo `__dict__`

Continuando con el ejemplo anterior:

```
>>> P1.__dict__
{'edad': 25, 'ciudad': 'Neira', 'nombre': 'Pepito Pérez', 'direccion':
'Carrera 2 # 3 - 42'}
>>> P2.__dict__
{'ciudad': 'Manizales', 'nombre': 'María Martínez'}
>>> Persona.__dict__
mappingproxy({'__doc__': None, '__weakref__': <attribute '__weakref__'
of 'Persona' objects>, '__dict__': <attribute '__dict__' of 'Persona'
objects>, 'edad': -1, 'ciudad': 'Aquí va la ciudad', 'nombre': 'Aquí v
a el nombre', 'direccion': 'Aquí va la dirección', '__module__': '__ma
in__'})
>>> |
```

- Con el atributo `__dict__` de una realización, se pueden observar los **atributos del objeto** junto con sus correspondientes valores.
- Con el atributo `__dict__` de una clase, se pueden observar los **atributos de la clase** junto con sus correspondientes valores.

# La función getattr()

```
>>> p = Perro()  
>>> getattr(p, 'color')  
'Negro'
```

```
>>> p.raza  
Traceback (most recent call last):
```

```
  File "<pyshell#43>", line 1, in <module>  
    p.raza
```

```
AttributeError: 'Perro' object has no attribute 'raza'
```

```
>>> getattr(p, 'raza')
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#44>", line 1, in <module>  
    getattr(p, 'raza')
```

```
AttributeError: 'Perro' object has no attribute 'raza'
```

```
>>> getattr(p, 'raza', 'Criollo')
```

```
'Criollo'
```

```
>>>
```

```
>>> help(getattr)
```

```
Help on built-in function getattr in module builtins:
```

```
getattr(...)
```

```
    getattr(object, name[, default]) -> value
```

Si el parámetro no existe, retorna el valor dado por defecto.

Get a named attribute from an object; `getattr(x, 'y')` is equivalent to `x.y`. When a default argument is given, it is returned when the attribute doesn't exist; without it, an exception is raised in that case.

# Agregando atributos a las funciones

```
>>> def f(x):
    return "Hola " + x
>>> f.y = 10
>>> print(f.y)
10
>>> f('Mario')
'Hola Mario'
```

Agregar atributos a objetos es posible en Python; incluso las funciones pueden tener atributos. Este tipo de atributos serían atributos del objeto.

```
def mifun(x):
    mifun.contador = getattr(mifun, 'contador', 0) + 1
    return "Hola {0}, mifun.contador = {1}".format(x, mifun.contador)

>>> for i in range(5): mifun('Pepito')

'Hola Pepito, mifun.contador = 1'
'Hola Pepito, mifun.contador = 2'
'Hola Pepito, mifun.contador = 3'
'Hola Pepito, mifun.contador = 4'
'Hola Pepito, mifun.contador = 5'
>>> print(mifun.contador)
```

5

Este hecho se puede utilizar para definir variables estáticas (en el sentido de lenguaje C) en Python. Las **variables estáticas** son aquellas que conservan su valor entre llamados a funciones

# El método `__str__()`

```
1  class Persona():
2      def __init__(self, nombre, edad):
3          self.nombre = nombre
4          self.edad = edad
5
6      def imprimir(self):
7          print('Llamando al método imprimir()')
8          print('Nombre =', self.nombre)
9          print('Edad =', self.edad, 'años')
10
11     def __str__(self):
12         cadena = 'Llamando al método __str__()\n' + \
13             'Nombre = {0}\n'.format(self.nombre) + \
14             'Edad = {0} años'.format(self.edad)
15
16
17 p = Persona('Pepito Pérez', 20)
18 p.imprimir()
19 print()
20 print(p)
```

Line: 11 of 24 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 13_metodo_str.py
Llamando al método imprimir()
Nombre = Pepito Pérez
Edad = 20 años
```

```
Llamando al método __str__()
Nombre = Pepito Pérez
Edad = 20 años
```

```
1  class Persona():
2      def __init__(self, nombre, edad):
3          self.nombre = nombre
4          self.edad = edad
5
6      def __str__(self):
7          cadena = 'Llamando al método __str__()\n' + \
8              'Nombre = {0}\n'.format(self.nombre) + \
9              'Edad = {0} años'.format(self.edad)
10
11
12 p = Persona('Pepito Pérez', 20)
13 print(p)
```

Line: 8 of 11 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 13_metodo_str.py
< main .Persona object at 0x7f41cf3611d0>
```

```
class Persona():
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
    def __str__(self):
        cadena = 'Llamando al método __str__()\n' + \
                 'Nombre = {0}\n'.format(self.nombre) + \
                 'Edad = {0} años'.format(self.edad)
        return cadena
    def __repr__(self):
        cadena = 'Llamando al método __repr__()\n' + \
                 'Nombre = {0}\n'.format(self.nombre) + \
                 'Edad = {0} años'.format(self.edad)
        return cadena
>>> p = Persona('Pepito Pérez', 20)
>>> p
Llamando al método __repr__()
Nombre = Pepito Pérez
Edad = 20 años
>>> repr(p)
'Llamando al método __repr__()\nNombre = Pepito Pérez\nEdad = 20 años'
>>> print(p)
Llamando al método __str__()
Nombre = Pepito Pérez
Edad = 20 años
>>> str(p)
'Llamando al método __str__()\nNombre = Pepito Pérez\nEdad = 20 años'
```

# Los métodos \_\_str\_\_() y \_\_repr\_\_()

```
>>> class A:  
    def __str__(self):  
        return "Hola desde __str__"  
  
>>> a = A()  
>>> a  
<__main__.A object at 0x7ff61ab12198>  
>>> repr(a)  
'<__main__.A object at 0x7ff61ab12198>'  
>>> str(a)  
'Hola desde __str__'  
>>> print(a)  
Hola desde __str__  
>>>  
>>> class B:  
    def __repr__(self):  
        return "Hola desde __repr__"  
  
>>> b = B()  
>>> b  
Hola desde __repr__  
>>> repr(b)  
'Hola desde __repr__'  
>>> str(b)  
'Hola desde __repr__'  
>>> print(b)  
Hola desde __repr__
```

If a class has a `__str__` method, the method will be used for an instance `x` of that class, if either the function `str` is applied to it or if it is used in a `print()` function. `__str__` will not be used, if `repr` is called, or if we try to output the value directly in an interactive Python shell.

Otherwise, if a class has only the `__repr__` method and no `__str__` method, `__repr__` will be applied in the situations, where `__str__` would be applied, if it were available.  
28

# Cuando usar `__repr__` y cuando usar `__str__`

```
>>> import datetime
>>> hoy = datetime.datetime.now()
>>> hoy
datetime.datetime(2015, 5, 19, 23, 26, 54,
362300)
>>> print(hoy)
2015-05-19 23:26:54.362300
>>>
>>> str_s = str(hoy)
>>> str_s
'2015-05-19 23:26:54.362300'
>>> repr_s = repr(hoy)
>>> repr_s
'datetime.datetime(2015, 5, 19, 23, 26, 54
, 362300)'
>>> eval(str_s)
Traceback (most recent call last):
File "<pyshell#24>", line 1, in <module>
    eval(str_s)
File "<string>", line 1
    2015-05-19 23:26:54.362300
    ^
SyntaxError: invalid token
>>> eval(repr_s)
datetime.datetime(2015, 5, 19, 23, 26, 54,
362300)
>>> hoy == eval(repr_s)
True
```

`__str__` is always the right choice, if the output should be for the end user or in other words, if it should be nicely printed.

`__repr__` on the other hand is used for the internal representation of an object. The output of `__repr__` should be, if feasible, a string which can be parsed by the python interpreter. The result of this parsing is an equal object.

This means that the following should be True for an object "obj":

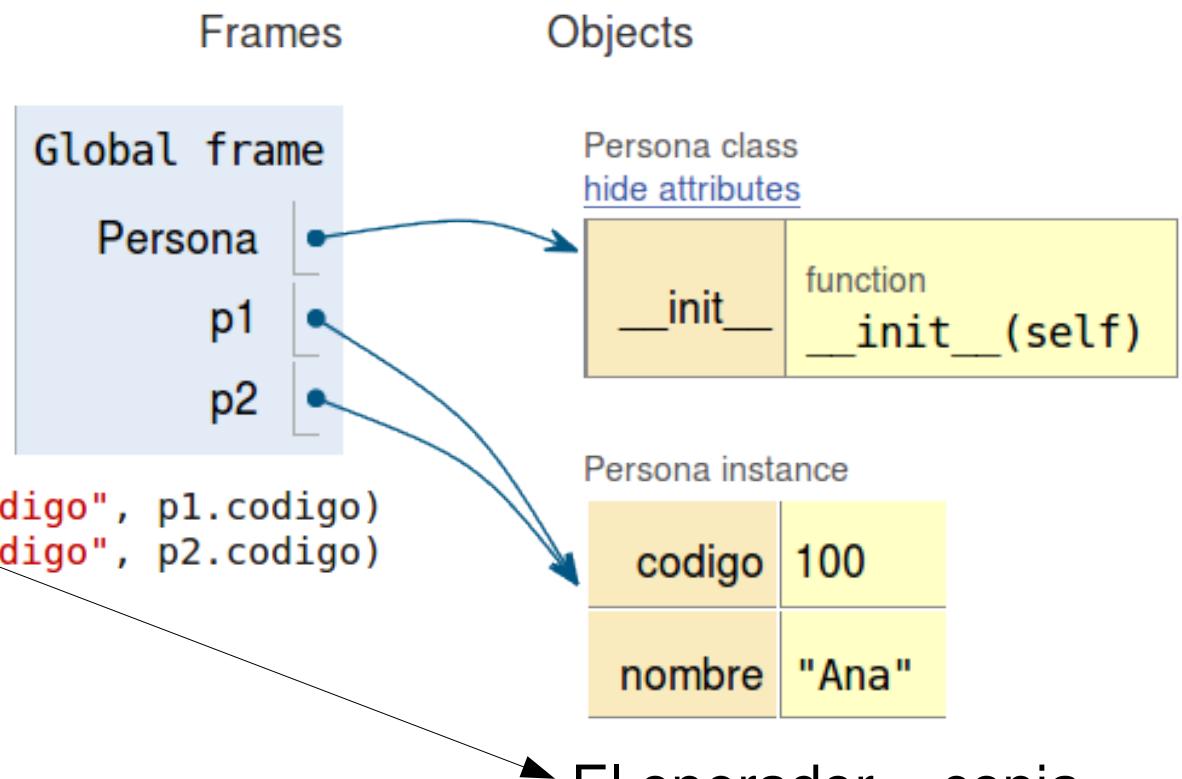
obj == eval(repr(obj))

# Referencias y objetos

```
1 class Persona:  
2     def __init__(self):  
3         self.nombre = ""  
4         self.codigo = 0  
5  
6     p1 = Persona()  
7     p1.nombre = "Juan"  
8     p1.codigo = 100  
9  
10    p2 = p1  
11    p2.nombre = "Ana"  
12  
13    print(p1.nombre, "tiene el código", p1.codigo)  
14    print(p2.nombre, "tiene el código", p2.codigo)  
15  
16    print(p1)  
17    print(p2)  
18    print(p1 is p2)  
19    print(type(p1))  
20    print(type(p2))
```

Line: 22 of 22 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 13_referencias.py  
Ana tiene el código 100  
Ana tiene el código 100  
<__main__.Persona object at 0x7f1c547b8f28>  
<__main__.Persona object at 0x7f1c547b8f28>  
True  
<class '__main__.Persona'>  
<class '__main__.Persona'>  
daalvarez@eredron:~ >
```



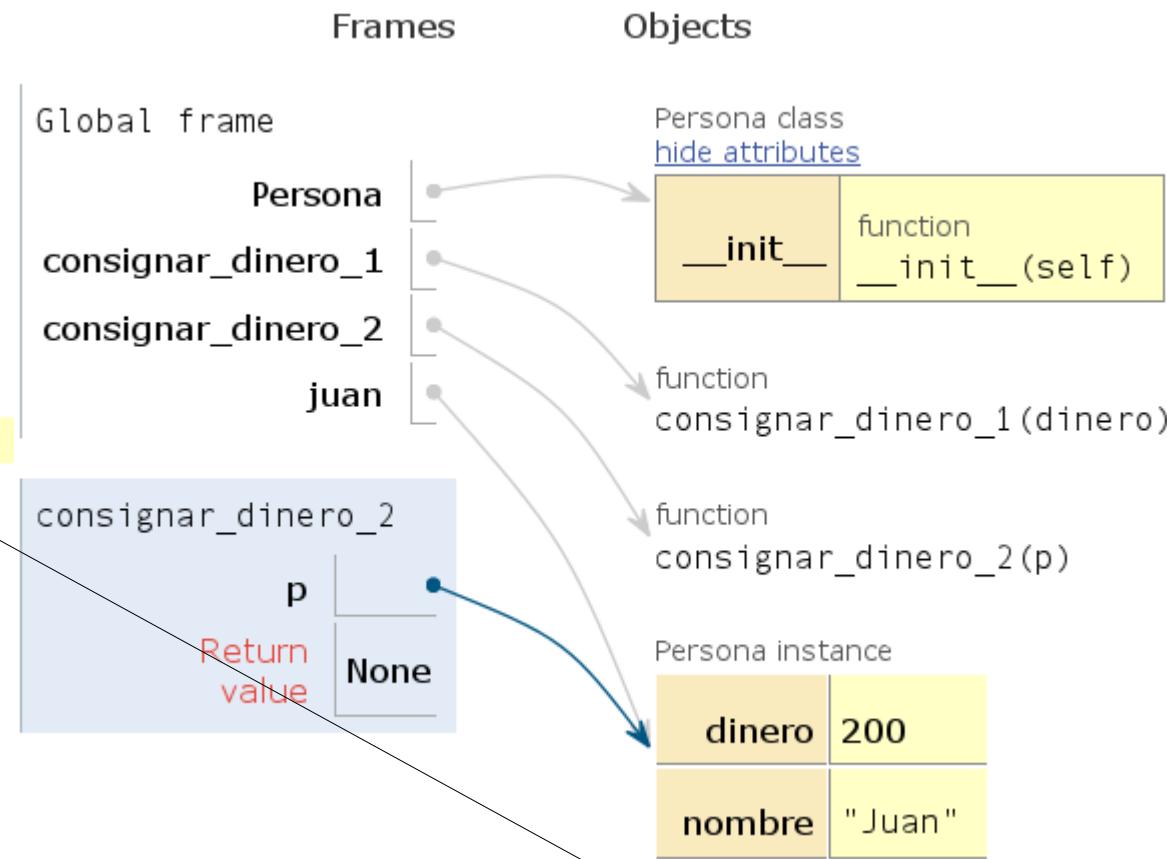
El operador = copia la referencia, no el objeto como tal.

# Objetos, funciones y referencias

```
1 ▼ class Persona:  
2     def __init__(self):  
3         self.nombre = ""  
4         self.dinero = 0  
5  
6     ▼ def consignar_dinero_1(dinero):  
7         dinero += 100  
8  
9     ▼ def consignar_dinero_2(p):  
10        p.dinero += 100  
11  
12    juan = Persona()  
13    juan.nombre = "Juan"  
14    juan.dinero = 100  
15  
16    print(juan.dinero)  
17  
18    consignar_dinero_1(juan.dinero)  
19    print(juan.dinero)  
20  
21    consignar_dinero_2(juan)  
22    print(juan.dinero)
```

Line: 10 of 24 Col: 1 LINE INS

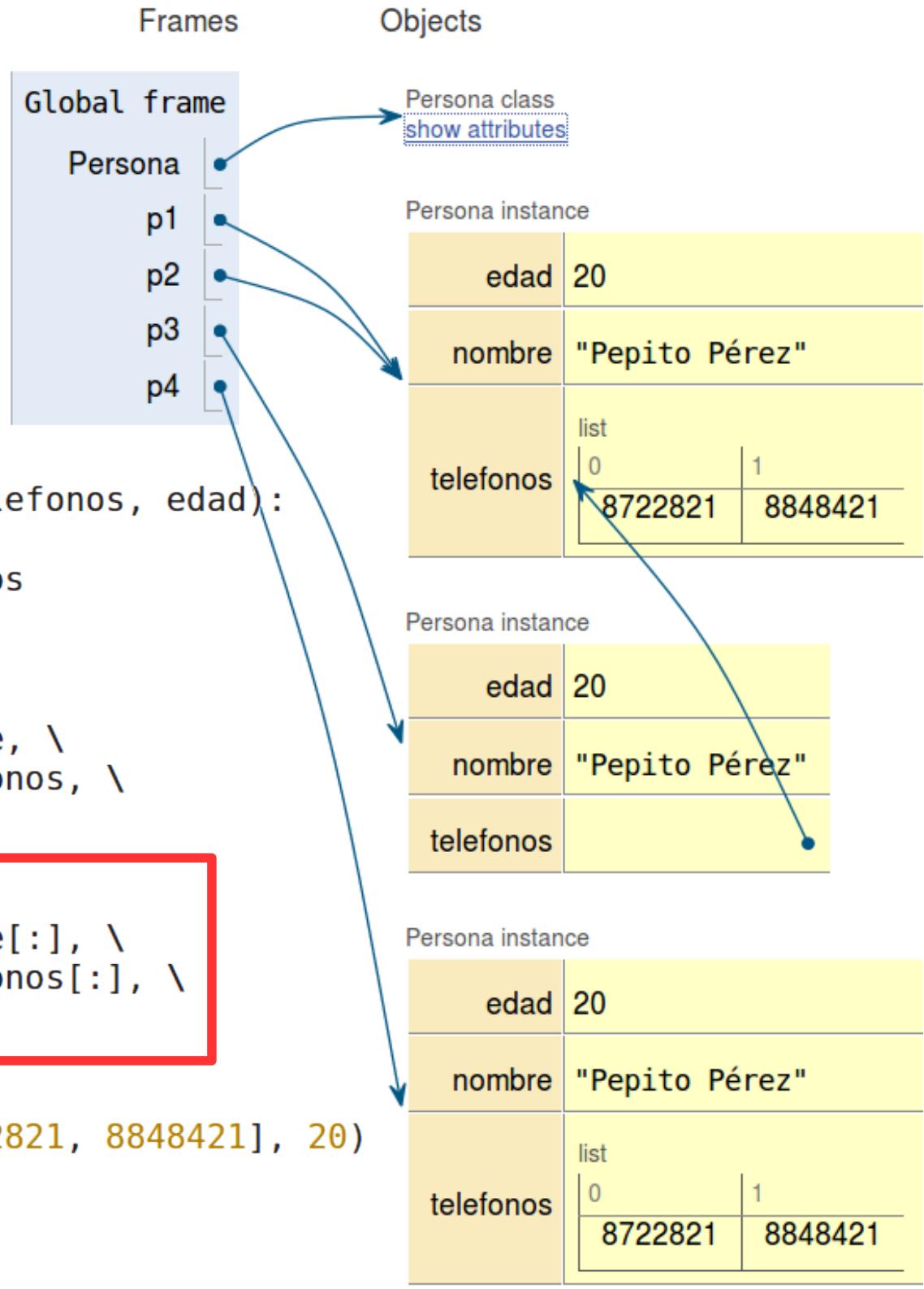
```
daa@heimdall ~ $ python3 13_fun_obj_ref.py  
100  
100  
200
```



Esta es la tabla de símbolos cuando se terminó de ejecutar esta línea. A `consignar_dinero_2()` se está transfiriendo una referencia a `juan`, no una copia de `juan`.

# El método de copia

```
1 class Persona():
2     def __init__(self, nombre, telefonos, edad):
3         self.nombre = nombre
4         self.telefonos = telefonos
5         self.edad = edad
6
7     def copia1(self):
8         return Persona(self.nombre, \
9                         self.telefonos, \
10                        self.edad)
11
12    def copia2(self):
13        return Persona(self.nombre[:], \
14                        self.telefonos[:], \
15                        self.edad)
16
17
18 p1 = Persona('Pepito Pérez', [8722821, 8848421], 20)
19 p2 = p1
20 p3 = p1.copia1()
21 p4 = p1.copia2()
```



# Atributos públicos, privados y protegidos

- Los atributos **privados** (comienzan con dos guiones bajos `__`) solo pueden ser llamados dentro de la definición de la clase. Por fuera de la clase son inaccesibles e invisibles.
- Los atributos **protegidos** o **restringidos** (comienzan con un guión bajo `_`) solo se pueden llamar desde las subclases cuando hay herencia y subclases.
- Los atributos **públicos** (comienzan con una letra) se pueden acceder libremente, dentro y fuera de la definición de la clase.

```
>>> class A:  
    def __init__(self):  
        self.__privado = 'Soy un atributo privado'  
        self._protegido = 'Soy un atributo protegido'  
        self.publico = 'Soy un atributo publico'
```

```
>>> x = A()  
>>> x.__privado  
Traceback (most recent call last):  
  File "<pyshell#45>", line 1, in <module>  
    x.__privado  
AttributeError: 'A' object has no attribute '__privado'  
>>> x._protegido  
'Soy un atributo protegido'  
>>> x.publico  
'Soy un atributo publico'  
>>>  
>>> x.publico += ' y me pueden cambiar'  
>>> x._protegido += ' y me pueden cambiar'  
>>> x.__privado += ' y me pueden cambiar'  
Traceback (most recent call last):  
  File "<pyshell#51>", line 1, in <module>  
    x.__privado += ' y me pueden cambiar'  
AttributeError: 'A' object has no attribute '__privado'  
>>>  
>>> x.__dict__  
{'_protegido': 'Soy un atributo protegido y me pueden cambiar',  
'publico': 'Soy un atributo publico y me pueden cambiar', '_A__p  
rivado': 'Soy un atributo privado'}
```

Se puede acceder (leer/escribir) a un objeto privado mediante:  
nombreclase\_atributo  
Por ejemplo \_A\_privado  
**SIN EMBARGO NO LO HAGA!**

```

1 # ATRIBUTOS PUBLICOS
2 ▼ class Cup:
3     def __init__(self):
4         self.color = None
5         self.content = None
6
7     def fill(self, beverage):
8         self.content = beverage
9
10    def empty(self):
11        self.content = None
12
13 redCup = Cup()
14 redCup.color = "red"
15 redCup.content = "tea"
16 redCup.empty()
17 redCup.fill("coffee")
18
19 # ATRIBUTOS PROTEGIDOS
20 ▼ class Cup:
21     def __init__(self):
22         self.color = None
23         self.__content = None # protected variable
24
25     def fill(self, beverage):
26         self.__content = beverage
27
28     def empty(self):
29         self.__content = None
30
31 redcup = Cup()
32 redcup.__content = "tea"

```

```

34 # ATRIBUTOS PRIVADOS
35 ▼ class Cup:
36     def __init__(self, color):
37         self._color = color # protected variable
38         self.__content = None # private variable
39
40     def fill(self, beverage):
41         self.__content = beverage
42
43     def empty(self):
44         self.__content = None
45
46 redCup = Cup("red")
47 redCup.__Cup__content = "tea"
48

```

Python doesn't have any mechanisms, that would effectively restrict you from accessing a variable or calling a member method. All of this is a matter of culture and convention. Por lo tanto, ¡siga las convenciones!

<http://radek.io/2011/07/21/private-protected-and-public-in-python/>

Observe que todo este código se ejecutó sin error alguno.

# El método destructor

In OOP, a **destructor** is a method which is automatically invoked when the object is disappears from memory. It can happen when its lifetime is bound to scope and the execution leaves the scope, when it is embedded into another object whose lifetime ends, or when it was allocated dynamically and is released explicitly. Its main purpose is to free the resources (memory allocations, open files or sockets, database connections, resource locks, etc.) which were acquired by the object along its life cycle and/or deregister from other entities which may keep references to it.

El destructor en Python es el método `del()`

# El método constructor `__init__()` y el método destructor `__del__()`

```
>>> class A:  
    def __init__(self, nombre):  
        self.__nombre = nombre  
        print(nombre, 'ha sido creado')  
    def __del__(self):  
        print(self.__nombre, 'ha sido destruido')  
  
>>> x = A('OBJETO 1')  
OBJETO 1 ha sido creado  
>>> y = A('OBJETO 2')  
OBJETO 2 ha sido creado  
>>> z = A('OBJETO 3')  
OBJETO 3 ha sido creado  
>>> z = y  
OBJETO 3 ha sido destruido → Se ejecutó cuando se hizo garbage collection  
>>> del z  
>>> del y  
OBJETO 2 ha sido destruido
```

```
class Robot:  
    __counter = 0  
  
    def __init__(self):  
        type(self).__counter += 1  
  
    def RobotInstances1(self):  
        return Robot.__counter  
  
    def RobotInstances2():  
        return Robot.__counter  
  
    @staticmethod  
    def RobotInstances3():  
        return Robot.__counter  
  
    >>> x = Robot()  
    >>> print(x.RobotInstances1())  
1  
    >>> y = Robot()  
    >>> print(x.RobotInstances1())  
2  
    >>> Robot.RobotInstances1()  
Traceback (most recent call last):  
  File "<pyshell#24>", line 1, in <module>  
    Robot.RobotInstances1()  
TypeError: RobotInstances1() missing 1 required positional argument: 'self'
```

# Métodos estáticos

Observe el siguiente problema cuando accedemos un atributo privado de la clase.

Si tratamos de utilizar el método Robot.RobotInstances1(), observe que aparece un error.

Este aparece ya que no existe un objeto asociado que se pueda referir a la variable **self**.

```
class Robot:  
    __counter = 0  
  
    def __init__(self):  
        type(self).__counter += 1  
  
    def RobotInstances1(self):  
        return Robot.__counter  
  
    def RobotInstances2():  
        return Robot.__counter  
  
    @staticmethod  
    def RobotInstances3():  
        return Robot.__counter  
  
>>> Robot.RobotInstances2()  
0  
>>> x = Robot()  
>>> print(x.RobotInstances2())  
Traceback (most recent call last):  
  File "<pyshell#31>", line 1, in <module>  
    print(x.RobotInstances2())  
TypeError: RobotInstances2() takes 0 positional arguments but 1 was given
```

Ahora, si tratamos de utilizar el método `RobotInstances2()`, el cual está definido sin el parámetro `self`, entonces podemos acceder al atributo privado `__counter` a través de `Robot.RobotInstances2()`; sin embargo, cuando lo tratamos de acceder a través de un objeto tenemos un error, porque el llamado del objeto inmediatamente pasa el parámetro `self` que nadie está recibiendo.

```

class Robot:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    def RobotInstances1(self):
        return Robot.__counter

    def RobotInstances2():
        return Robot.__counter

    @staticmethod
    def RobotInstances3():
        return Robot.__counter

>>> Robot.RobotInstances3()
0
>>> x = Robot()
>>> print(x.RobotInstances3())
1
>>> y = Robot()
>>> print(y.RobotInstances3())
2
>>> Robot.RobotInstances3()
2

```

Nosotros queremos un método que pueda acceder al atributo privado de la clase ya sea a través del nombre de la clase o a través del nombre de un objeto.

La solución la proveen los llamados **métodos estáticos**, los cuales no requieren una referencia a un objeto.

Para crear un método estático se requiere agregar la línea `@staticmethod` justo antes del encabezado del método. La línea `@staticmethod` se le conoce en la terminología de Python como un **decorador**.

Observe entonces el comportamiento del método estático `RobotInstances3()`. Es justo lo que deseábamos.

# Métodos clase

```
class Robot:  
    __counter = 0  
  
    def __init__(self):  
        type(self).__counter += 1  
  
    @classmethod  
    def RobotInstances(ref_clase):  
        return ref_clase, Robot.__counter  
  
=>  
if __name__ == "__main__":  
    print(Robot.RobotInstances())  
    x = Robot()  
    print(x.RobotInstances())  
    y = Robot()  
    print(x.RobotInstances())  
    print(Robot.RobotInstances())  
  
<class '__main__.Robot', 0>  
<class '__main__.Robot', 1>  
<class '__main__.Robot', 2>  
<class '__main__.Robot', 2>  
=> |
```

Los **métodos clase** (class methods), al igual que los métodos estáticos no dependen de los objetos, pero a diferencia de los métodos estáticos, ellos si dependen de la clase a la cual pertenecen. De hecho, el primer parámetro de un método clase es una referencia a la clase.

Estos métodos se pueden llamar a través de un objeto o utilizando el nombre de la clase.

# Métodos estáticos y métodos clase

```
1  class fraccionario:
2
3  def __init__(self, n, d):
4      self.numerador, self.denominador = fraccionario.simplificar(n, d)
5
6  @staticmethod
7  def maximo_comun_divisor(a, b):
8      while b != 0:
9          a, b = b, a%b
10     return a
11
12 @classmethod
13 def simplificar(cls, n1, n2):
14     g = cls.maximo_comun_divisor(n1, n2)
15     return (n1 // g, n2 // g)
16
17 def __str__(self):
18     return str(self.numerador) + '/' + str(self.denominador)
19
20 x = fraccionario(8,24)
21 print(x)
```

Line 22, Column 1

```
daalvarez@eredron ~ $ python3 13_fraccionario.py
1/3
daalvarez@eredron ~ $ □
```

# Métodos estáticos vs Métodos clase

```
1  ▼ class Mascotas1:  
2      nombre = "mascoticas"  
3  
4      @staticmethod  
5      ▼ def acerca_de():  
6          print("Esta clase es sobre {}.".format(Mascotas1.nombre))  
7  
8  
9  
10     ▼ class Perros1(Mascotas1):  
11         nombre = "perritos"  
12  
13     ▼ class Gatos1(Mascotas1):  
14         nombre = "gaticos"  
15  
16     p = Mascotas1()  
17     p.acerca_de()  
18     d = Perros1()  
19     d.acerca_de()  
20     c = Gatos1()  
21     c.acerca_de()  
22  
23     print(80*'-')
```

```
25    ▼ class Mascotas2:  
26        nombre = "mascoticas"  
27  
28        @classmethod  
29        ▼ def acerca_de(cls):  
30            print("Esta clase es sobre {}.".format(cls.nombre))  
31  
32  
33  
34    ▼ class Perros2(Mascotas2):  
35        nombre = "perritos"  
36  
37    ▼ class Gatos2(Mascotas2):  
38        nombre = "gaticos"  
39  
40    p = Mascotas2()  
41    p.acerca_de()  
42    d = Perros2()  
43    d.acerca_de()  
44    c = Gatos2()  
45    c.acerca_de()
```

Line 46, Column 1

```
daalvarez@eredron ~ $ python3 13_staticmethods_classmethods.py  
Esta clase es sobre mascoticas.  
Esta clase es sobre mascoticas.  
Esta clase es sobre mascoticas.  
-----  
Esta clase es sobre mascoticas.  
Esta clase es sobre perritos.  
Esta clase es sobre gaticos.  
daalvarez@eredron ~ $ █
```

# Encapsulación de datos

[http://en.wikipedia.org/wiki/Encapsulation\\_%28object-oriented\\_programming%29](http://en.wikipedia.org/wiki/Encapsulation_%28object-oriented_programming%29)

En POO, se denomina **encapsulamiento** al ocultamiento de los atributos y/o métodos de un objeto de manera que solo se pueda cambiar mediante las métodos definidos para esa clase.

El aislamiento protege a los datos asociados de un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones. De esta forma el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos. Por otro lado se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas.

Formas de encapsular: Pública, Protegida o restringida, Privada

# Métodos de acceso y métodos de mutación (getters and setters)

[http://en.wikipedia.org/wiki/Mutator\\_method](http://en.wikipedia.org/wiki/Mutator_method)

```
class P:  
    def __init__(self,x):  
        self.__x = x  
  
    def getX(self): → Método de acceso  
        return self.__x  
  
    def setX(self, x): → Método de mutación  
        self.__x = x
```

```
>>> p1 = P(42)  
>>> p2 = P(4711)  
>>> p1.getX()  
42  
>>> p1.setX(47)  
>>> p1.setX(p1.getX() + p2.getX())  
>>> p1.getX()  
4758
```

Sin embargo hubiera sido deseable escribir  
**p1.x = p1.x + p2.x**

ya que en este caso no se justifica usar los métodos `getX()` y `setX()` y hacer encapsulación de datos. No se puede hacer porque “`x`” es privada.

Estos métodos sirven para implementar el principio de la **encapsulación de datos** en POO.

- Los **métodos de acceso** (**getters** o **accessors** en inglés), sirven para acceder a las propiedades (privadas) de un objeto.
- Los **métodos de mutación** (**setters** o **mutators** en inglés), sirven para cambiar las propiedades (privadas) de un objeto.

En este caso sí se justifica crear ambos métodos:

```
class P:  
    def __init__(self,x):  
        self.setX(x)  
  
    def getX(self):  
        return self.__x  
  
    def setX(self, x):  
        if x < 0:  
            self.__x = 0  
        elif x > 1000:  
            self.__x = 1000  
        else:  
            self.__x = x  
  
  
=>> p1 = P(1001)  
=>> p1.getX()  
1000  
=>> p2 = P(15)  
=>> p2.getX()  
15  
=>> p3 = P(-1)  
=>> p3.getX()  
0
```

Sin embargo, Python ofrece una solución más elegante:

```
class P:  
    def __init__(self,x):  
        self.x = x  
  
    @property # esto se llama un "decorador"  
    def x(self): # este es el getter  
        return self.__x  
  
    @x.setter # esto se llama un "decorador"  
    def x(self, x): # este es el setter  
        if x < 0:  
            self.__x = 0  
        elif x > 1000:  
            self.__x = 1000  
        else:  
            self.__x = x  
  
  
=>> p1 = P(500)  
=>> p1.x  
500  
=>> p1.x = 2000  
=>> p1.x  
1000  
=>> p1.x = -100  
=>> p1.x  
0  
=>> p1.x = 123  
=>> p1.x  
123
```

```

class P:
    def __init__(self,x):
        self.x = x

    @property # esto se llama un "decorador"
    def x(self): # este es el getter
        return self.__x

    @x.setter # esto se llama un "decorador"
    def x(self, x): # este es el setter
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x

>>> p1 = P(500)
>>> p1.x
500
>>> p1.x = 2000
>>> p1.x
1000
>>> p1.x = -100
>>> p1.x
0
>>> p1.x = 123
>>> p1.x
123

```

Dos observaciones importantes:

- We just put the code line "self.x = x" in the `__init__` method and the property method `x` is used to check the limits of the values.
- We wrote "two" methods with the same name and a different number of parameters "`def x(self)`" and "`def x(self,x)`". Esto se supone que no es posible, pero con el uso del decorador, se permite la sobrecarga de funciones.

```
>>> class P:  
    def __init__(self,x):  
        self.__set_x(x)  
  
    def __get_x(self):      # getter (es un método privado)  
        return self.__x  
  
    def __set_x(self, x): # setter (es un método privado)  
        if x < 0:  
            self.__x = 0  
        elif x > 1000:  
            self.__x = 1000  
        else:  
            self.__x = x  
  
    x = property(__get_x, __set_x)
```

```
>>> p1 = P(500)           I  
>>> p1.x  
500  
>>> p1.x = 2000  
>>> p1.x  
1000  
>>> p1.x = -100  
>>> p1.x  
0  
>>> p1.x = 123  
>>> p1.x  
123
```

## Forma alternativa para definir los getters y setters

Sin embargo se prefiere utilizar  
la forma que usa los decoradores

```

class Robot:
    def __init__(self, name, build_year, lk = 0.5, lp = 0.5 ):
        self.name = name
        self.build_year = build_year
        self.__potential_physical = lk
        self.__potential_psychic = lp

    @property
    def condition(self):
        s = self.__potential_physical + self.__potential_psychic
        if s <= -1:
            return "I feel miserable!"
        elif s <= 0:
            return "I feel bad!"
        elif s <= 0.5:
            return "Could be worse!"
        elif s <= 1:
            return "Seems to be okay!"
        else:
            return "Great!"
```

En conclusión, la mejor forma de acceder a los atributos de una clase es usando decoradores, en caso de ser necesario.

```

>>> x = Robot("Marvin", 1979, 0.2, 0.4 )
>>> y = Robot("Caliban", 1993, -0.4, 0.3)
>>> print(x.condition)
Seems to be okay!
>>> print(y.condition)
I feel bad!
>>>
```

# Los métodos también se pueden hacer privados

```
>>> class P:  
    __x = 10  
    def __printx(self):-----> Método privado  
        print(self.__x)  
    def printx1(self):  
        print(self.__x)  
    def printx2(self):  
        self.__printx()  
  
>>> obj = P()  
>>> obj.printx1()  
10  
>>> obj.printx2()  
10  
>>> obj.__printx()  
Traceback (most recent call last):  
  File "<pyshell#87>", line 1, in <module>  
    obj.__printx()  
AttributeError: 'P' object has no attribute '__printx'
```

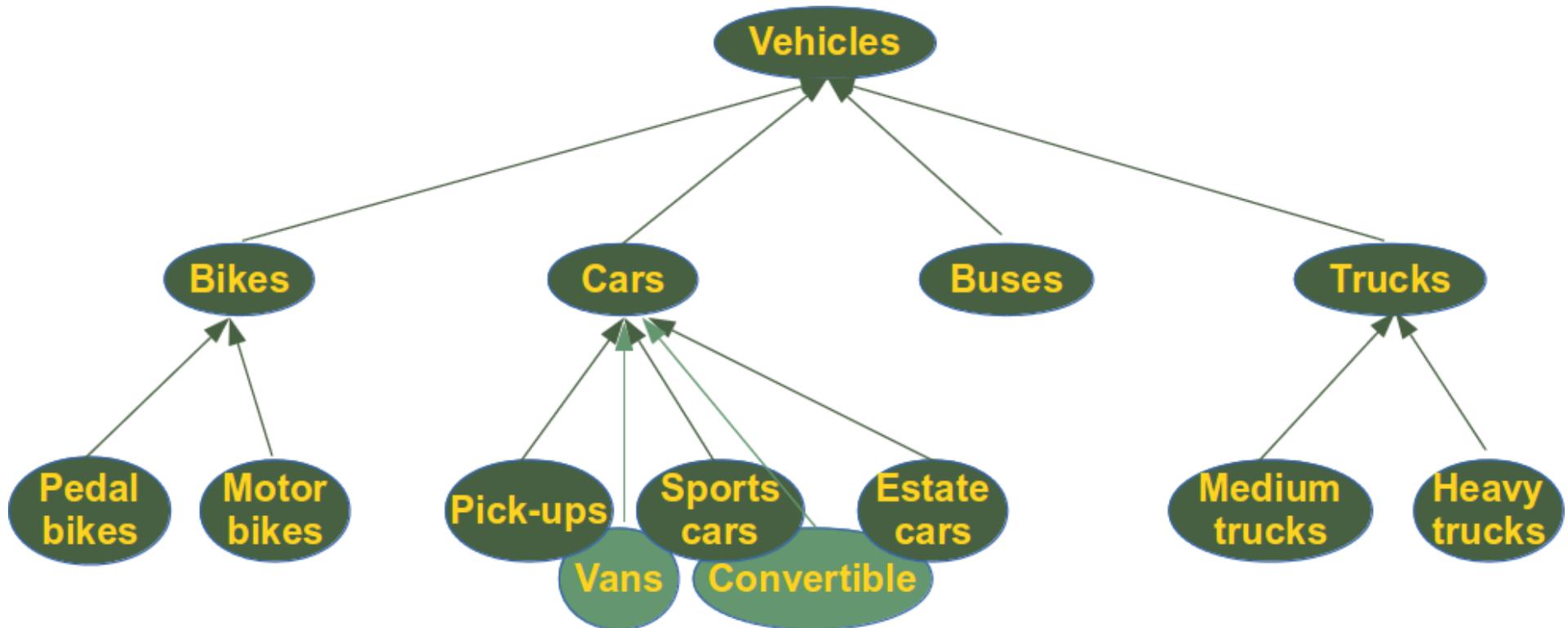
# Herencia

- Another powerful feature of using classes and objects is the ability to make use of inheritance. It is possible to create a class and inherit all of the attributes and methods of a parent class.
- Our program can create child classes that will inherit all the attributes and methods of the parent class. The child classes may then add fields and methods that correspond to their needs.

# Herencia

A class can inherit attributes and behaviour methods from another class, called the superclass. A class which inherits from a superclass is called a subclass, also called heir class or child class. Superclasses are sometimes called ancestors as well. There exists a hierarchy relationship between classes. It's similar to relationships or categorizations that we know from real life.

# Herencia



Bikes, cars, buses and trucks are vehicles. pick-ups, vans, sports cars, convertibles and estate cars are all cars and by being cars they are vehicles as well. We could implement a vehicle class in Python, which might have methods like accelerate and brake. Cars, Buses and Trucks and Bikes can be implemented as subclasses which will inherit these methods from vehicle.

```

1 ▼ class Persona:
2     nombre = ""
3     def __init__(self, nompers):
4         self.nombre = nompers
5         print("Persona creada")
6     def imprimir(self):
7         print(self.nombre)
8
9 ▼ class Empleado(Persona):
10    nombre_del_puesto = ""
11    def imprimir(self):
12        print(self.nombre, self.nombre_del_puesto)
13
14 ▼ class Cliente(Persona):
15    email = ""
16    def imprimir(self):
17        print(self.nombre, self.email)
18
19 johnSmith = Persona("John Smith")
20 johnSmith.imprimir()
21 janeEmpleado = Empleado("Jane Empleado")
22 janeEmpleado.nombre_del_puesto = "Desarrollador Web"
23 janeEmpleado.imprimir()
24 bobCliente = Cliente("Bob Cliente")
25 bobCliente.email = "enviame@spam.com"
26 bobCliente.imprimir()

```

Line: 28 of 28 Col: 1 LINE INS

daa@heimdall ~ \$ python3 13\_herencia.py

Persona creada

John Smith

Persona creada

Jane Empleado Desarrollador Web

Persona creada

Bob Cliente enviame@spam.com

Aquí el atributo nombre se está heredando.

Observe que los métodos también se heredan. En este caso, el constructor se está heredando.

Cuando la clase hija define un método con el mismo nombre que la clase padre, el método de la clase padre se reemplaza por el método de la clase hija.

```

1  class Persona:
2      nombre = ''
3      def __init__(self, nompers):
4          self.nombre = nompers
5          print("Persona creada")
6
7      def __str__(self):
8          return self.nombre
9
10 class Empleado(Persona):
11     nombre_del_cargo = ''
12     def __init__(self, nompers='', nomcargo=''):
13         self.nombre_del_cargo = nomcargo
14         super().__init__(nompers)
15         print("Empleado creado")
16
17     def __str__(self):
18         return super().__str__() + ' ' + self.nombre_del_cargo
19
20
21 johnSmith = Persona("John Smith")
22 print(johnSmith)
23
24 janeEmpleado = Empleado("Jane Empleado", "Abogada")
25 print(janeEmpleado)
26

```

En Python 2 sería:

`super(Empleado, self).__init__(nompers)`  
Python 3 también soporta esa construcción

Si se desea ejecutar ambos constructores, el de la clase padre y el de la clase hija, la clase hija debe llamar explícitamente el constructor de la clase padre.

Adicionalmente, se está **anteponiendo** el método `Empleado.imprimir()` al método `Persona.imprimir()`

Line 14, Column 1

daalvarez@eredron ~ \$ python3 13\_herencia\_3.py

Persona creada

John Smith

Persona creada

Empleado creado

Jane Empleado Abogada

daalvarez@eredron ~ \$

```

1  class Persona:
2      nombre = ''
3      def __init__(self, nompers):
4          self.nombre = nompers
5          print('Persona creada')
6
7      def __str__(self):
8          return self.nombre
9
10 class Empleado(Persona):
11     nombre_del_cargo = ''
12     def __init__(self, nompers='', nomcargo=''):
13         self.nombre_del_cargo = nomcargo
14         # Persona.__init__(nompers) -> PRODUCE ERROR
15         Persona.__init__(self, nompers)
16         print('Empleado creado')
17
18     def __str__(self):
19         return Persona.__str__(self) + ' ' + self.nombre_del_cargo
20
21
22 johnSmith = Persona('John Smith')
23 print(johnSmith)
24
25 janeEmpleado = Empleado('Jane Empleado', 'Abogada')
26 print(janeEmpleado)

```

Otra forma como se pueden llamar los métodos de la clase padre desde la clase hija:

Line 15, Column 1

daalvarez@eredron ~ \$ python3 13\_herencia\_2.py

Persona creada

John Smith

Persona creada

Empleado creado

Jane Empleado Abogada

daalvarez@eredron ~ \$

# Métodos públicos, privados y protegidos + herencia

# Sobrecarga de funciones

In the context of object-oriented programming, **overloading** is the ability to define the same method, with the same name but with a different number of arguments and types. It's the ability of one function to perform different tasks, depending on the number of parameters or the types of the parameters.

[http://www.python-course.eu/python3\\_inheritance.php](http://www.python-course.eu/python3_inheritance.php)

# Herencia múltiple

- Using multiple inheritance, a class can inherit attributes and methods from more than one parent class.

## Más métodos especiales

Hemos visto que las clases admiten dos métodos especiales: `__init__` y `__str__`. No son los únicos métodos especiales. Podemos hacer que las clases se comporten de modo similar a los tipos de datos nativos de Python definiendo muchos otros métodos especiales. He aquí unos pocos:

- `__len__(self)`: Permite aplicar la función predefinida `len` sobre objetos de la clase. Debe devolver la «longitud» o «talla» del objeto. En el caso de colas y conjuntos, por ejemplo, correspondería al número de elementos. Si  $A$  es un *Conjunto*, podríamos usar `len(A)` si antes hubiésemos definido el método `__len__`.
- `__add__(self, otro)`: Permite aplicar el operador de suma (+) a objetos de la clase sobre la que se ha definido. Si, por ejemplo,  $A$  y  $B$  son conjuntos, la expresión  $C = A + B$  permite asignar al nuevo conjunto  $C$  la unión de ambos.
- `__mul__(self, otro)`: Permite aplicar el operador de multiplicación (\*) a objetos de la clase sobre la que se ha definido.
- `__cmp__(self, otro)`: Permite aplicar los operadores de comparación (`<`, `>`, `<=`, `>=`, `==`, `!=`) a objetos de una clase. Debe devolver -1 si `self` es menor que `otro`, 0 si son iguales y 1 si `self` es mayor que `otro`.

Podemos, por ejemplo, definir `__cmp__` en *Persona* para que devuelva -1 cuando la edad `self.edad` es menor que `otro.edad`, 0 si son iguales y 1 si `self.edad` es mayor que `otro.edad`. Si *juan* y *pedro* son personas, podremos compararlas con expresiones como *juan < pedro* o *juan != pedro*.

# En resumen, los principios básicos de la POO son:

- Encapsulación
- Abstracción de datos
- Polimorfismo
- Herencia
- Sobrecarga de operadores

# Referencias

- Wikipedia
- <http://www.programarcadegames.com/>
- [http://www.python-course.eu/python3\\_course.php](http://www.python-course.eu/python3_course.php)
- Documentación de Python:
  - <https://docs.python.org/3/tutorial/index.html>
  - <https://docs.python.org/3/>
- Marzal Varó, Andrés; Gracia Luengo, Isabel; García Sevilla, Pedro (2014). Introducción a la programación con Python 3. Disponible en: <http://dx.doi.org/10.6035/Sapientia93>

# Sobrecarga de funciones

[http://en.wikipedia.org/wiki/Function\\_overloading](http://en.wikipedia.org/wiki/Function_overloading)

Sobrecarga es la capacidad de un lenguaje de programación, que permite nombrar con el mismo identificador diferentes variables u operaciones.

En POO la sobrecarga se refiere a la posibilidad de tener dos o más funciones con el mismo nombre pero funcionalidad diferente. Es decir, dos o más funciones con el mismo nombre realizan acciones diferentes. Esto ocurre porque las funciones reciben un número variado de argumentos o las funciones reciben diferentes tipos de datos.

# Funciones sobrecargadas para soportar diferentes tipos de datos

```
>>> def successor(number):
    return number + 1

>>> successor(1)
2
>>> successor(1.6)
2.6
>>> successor([1,2,3])
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    successor([1,2,3])
  File "<pyshell#2>", line 2, in successor
    return number + 1
TypeError: can only concatenate list (not "int") to list
>>> |
```

The screenshot shows a terminal window with the following content:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int successor(int number)
6 {
7     return number + 1;
8 }
9
10 double successor(double number)
11 {
12     return number + 1;
13 }
14
15 int main() {
16     cout << successor(10) << endl;
17     cout << successor(10.3) << endl;
18     return 0;
19 }
```

Line: 21 of 21 Col: 1 LINE INS

```
daalvarez@eredron:~ > g++ 13_sobrecarga1.cpp
daalvarez@eredron:~ > ./a.out
11
11.3
daalvarez@eredron:~ > █
```

# Funciones sobrecargadas para soportar un número variable de argumentos

```
>>> def f(n):
    return n+42

>>> def f(n,m):
    return n+m+42

>>> f(3,4)
49
>>> f(3)
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    f(3)
TypeError: f() missing 1 required positional argument: 'm'
>>> |
```

```
1 #include <iostream>
2
3 using namespace std;
4
5 int f(int n);
6 int f(int n, int m);
7
8 int main(void)
9 {
10     cout << "f(3): " << f(3) << endl;
11     cout << "f(3, 4): " << f(3, 4) << endl;
12     return 0;
13 }
14
15 int f(int n)
16 {
17     return n + 42;
18 }
19
20 int f(int n, int m)
21 {
22     return n + m + 42;
23 }
```

```
Line: 25 of 25 Col: 1      LINE  INS
daalvarez@eredron:~ > g++ 13_sobrecarga2.cpp
daalvarez@eredron:~ > ./a.out
f(3): 45
f(3, 4): 49
daalvarez@eredron:~ > █
```

```

1 #include <iostream>
2
3 using namespace std;
4
5 int f(int n);
6 int f(int n, int m);
7
8 int main(void)
9 {
10     cout << "f(3): " << f(3) << endl;
11     cout << "f(3, 4): " << f(3, 4) << endl;
12     return 0;
13 }
14
15 int f(int n)
16 {
17     return n + 42;
18 }
19
20 int f(int n, int m)
21 {
22     return n + m + 42;
23 }
```

Line: 25 of 25 Col: 1 LINE INS

```

daalvarez@eredron:~ > g++ 13_sobrecarga2.cpp
daalvarez@eredron:~ > ./a.out
f(3): 45
f(3, 4): 49
daalvarez@eredron:~ >
```

```

>>> def f(n, m=None):
        if m:
            return n+m+42
        else:
            return n+42

>>> f(3,4)
49
>>> f(3)
45
>>> |
```

```

>>> def f(*x):
        if len(x) == 1:
            return x[0]+42
        else:
            return x[0]+x[1]+42

>>> f(3,4)
49
>>> f(3)
45
>>> |
```

The \* operator can be used as a more general approach for a family of functions with 1, 2, 3, or even more parameters