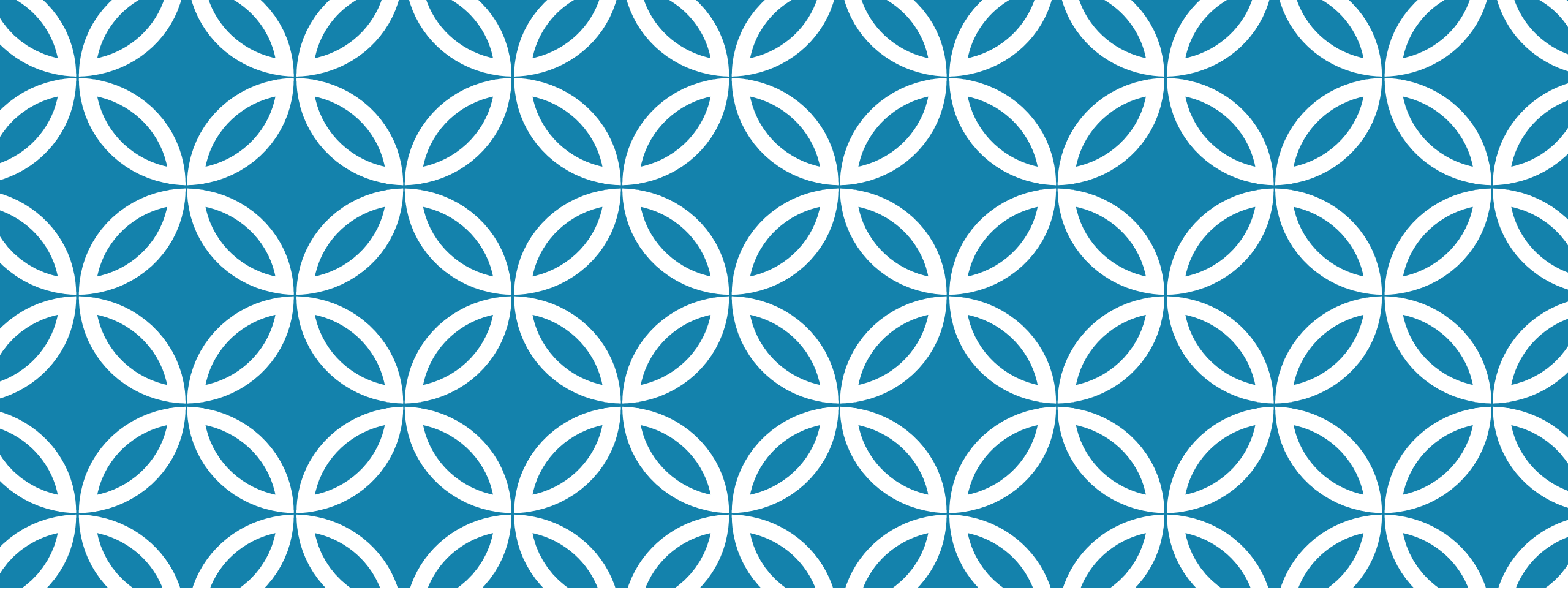


PROGRAMACIÓN ORIENTADA A OBJETOS

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**"). Evite confundir una clase con un objeto. Jorge y María son instancias de la clase “Persona”. Nombre la clase con la primera letra 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()).

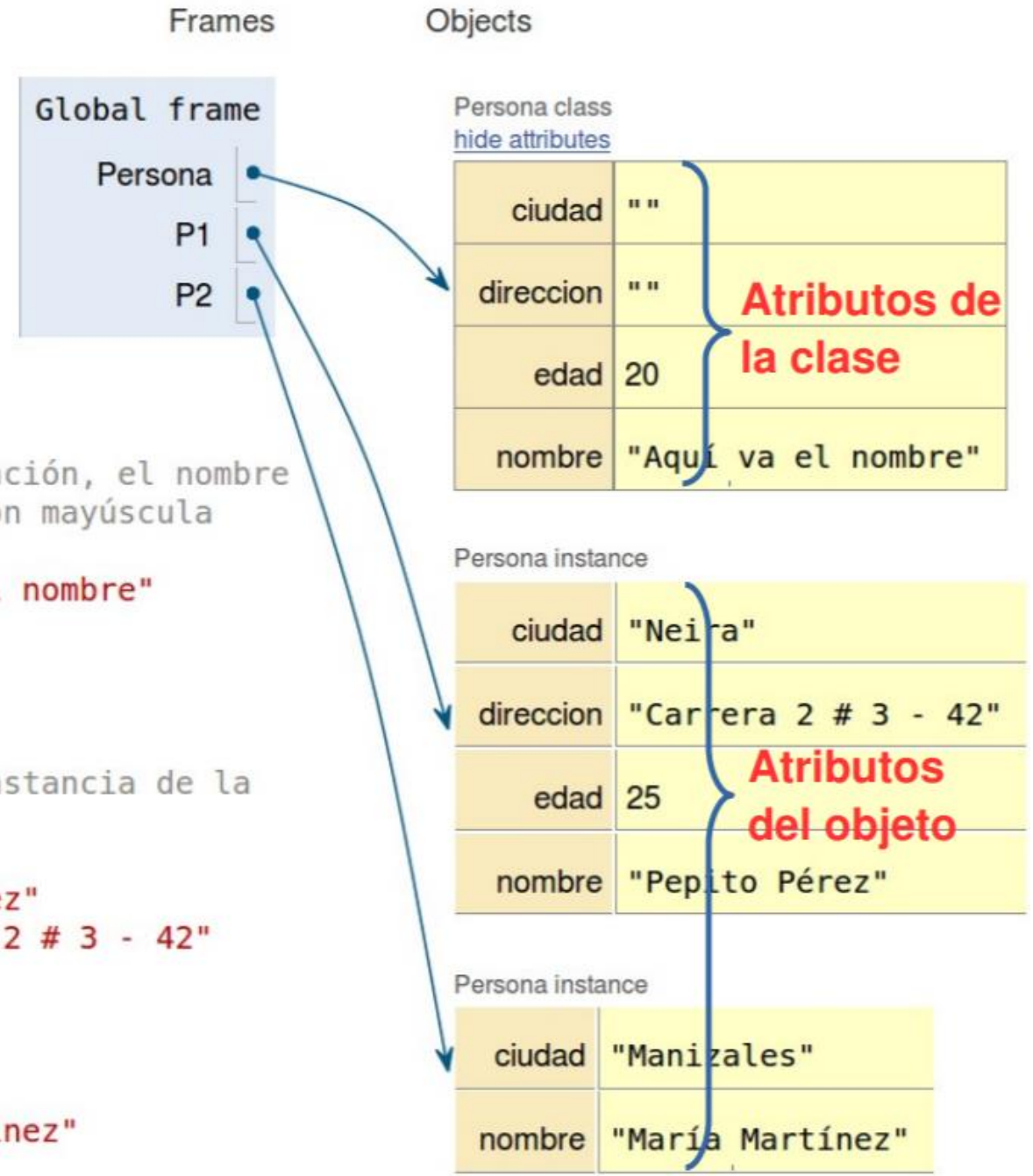
La clase mínima en Python

```
>>> 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
```

Las dos instancias de la clase Robot son diferentes.

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

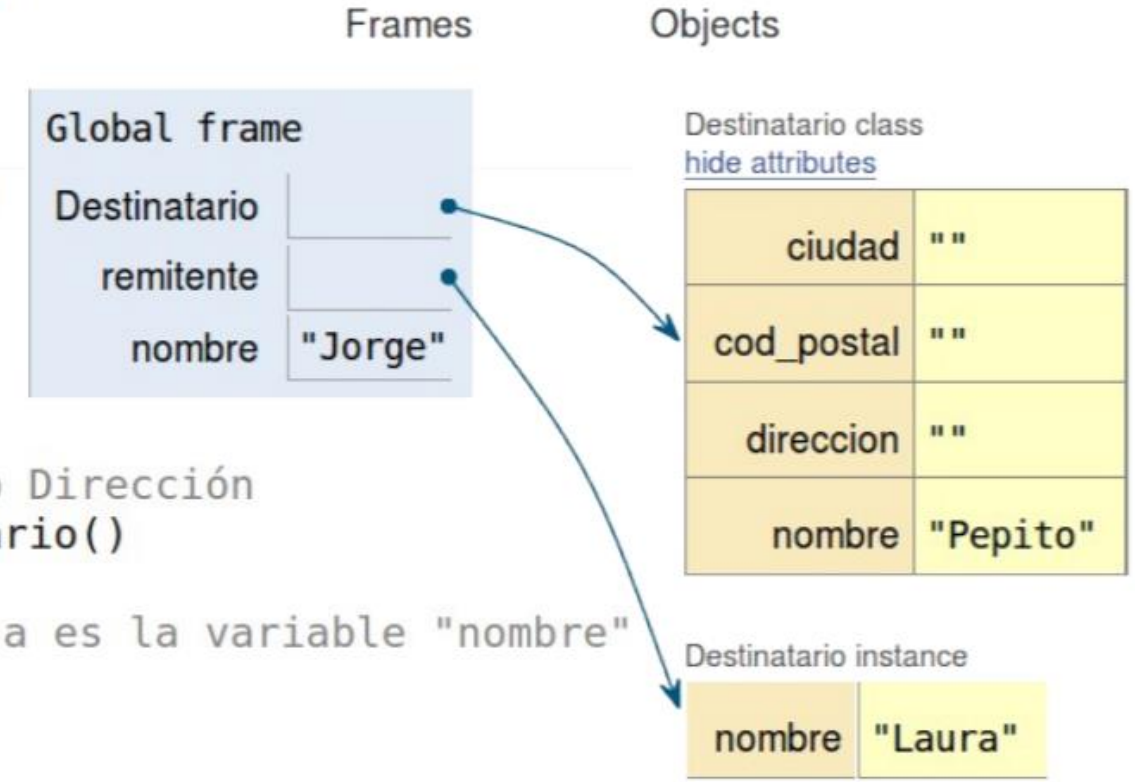
Clases de solo atributos (similar a las estructuras en C/C++)



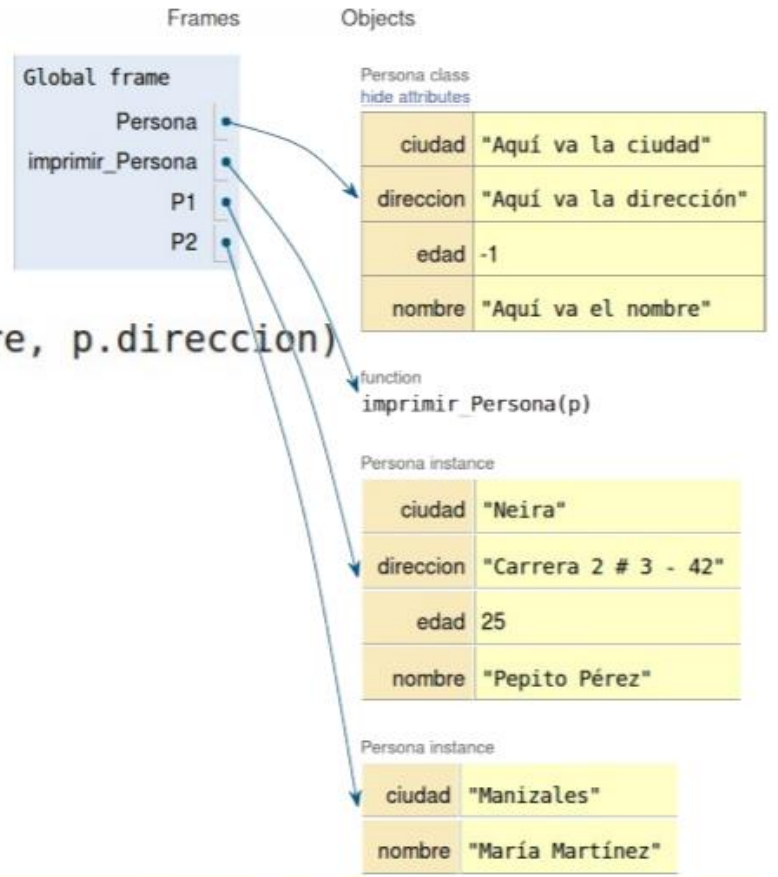

```
1  # Observe que por convención, el nombre
2  # de la clase empieza con mayúscula
3  class Persona():
4      nombre = "Aquí va el nombre"
5      direccion = ""
6      edad = 20
7      ciudad = ""
8
9  # Se crea un objeto o instancia de la
10 # clase Persona()
11 P1 = Persona()
12 P1.nombre = "Pepito 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 instancia 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"
```



```python
class Persona():
    nombre = "Aquí va el nombre"
    direccion = "Aquí va la dirección"
    edad = -1
    ciudad = "Aquí va la ciudad"

def imprimir_Persona(p):
    print('Nombre y dirección =', p.nombre, p.direccion)
    # Si existe una edad, imprimirla
    if p.edad >= 0:
        print('Edad =', p.edad)
    # Si existe una ciudad, imprimirla
    if p.ciudad != '':
        print('Ciudad =', p.ciudad)

P1 = Persona()
P1.nombre = "Pepito Pérez"
P1.direccion = "Carrera 2 # 3 - 42"
P1.ciudad = "Neira"
P1.edad = 25

P2 = Persona()
P2.nombre = "María Martíne
P2.ciudad = "Manizales"

imprimir_Persona(P1)
imprimir_Persona(P2)

print("El nombre P1 es", P1.nombre)
print("La dirección de P2 es", P2.direccion)
```



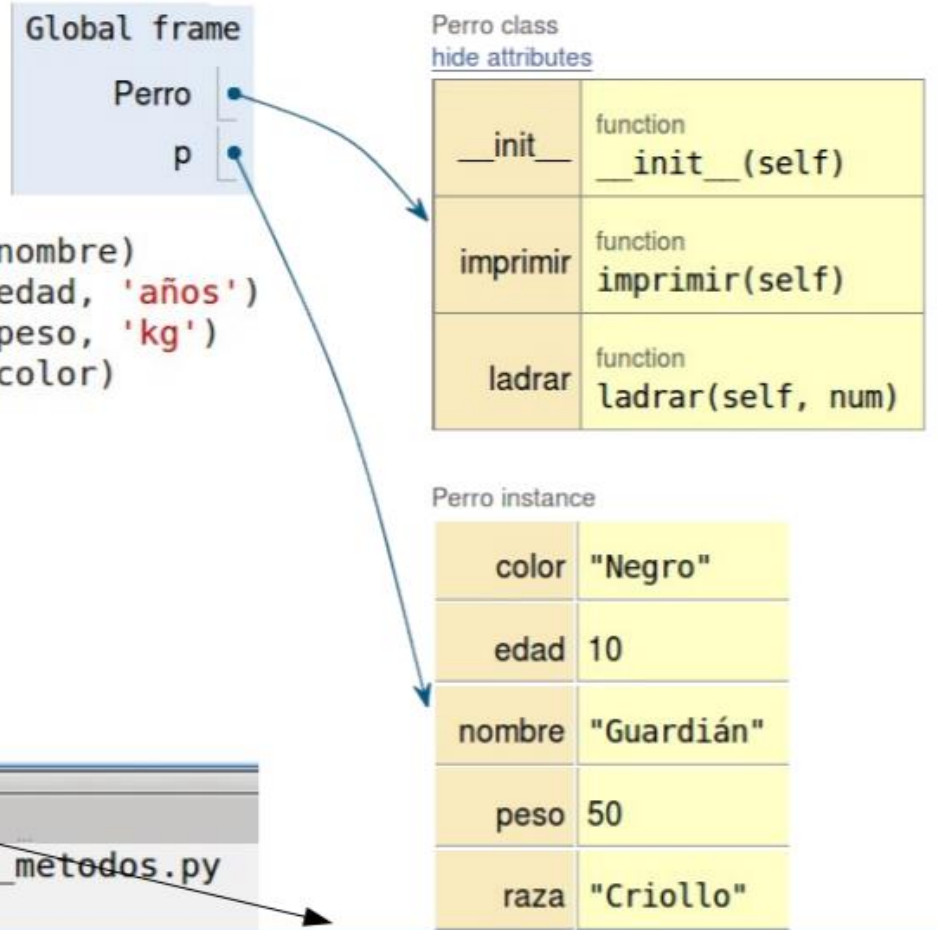

```
daalvarez@eredron:~ > python3 13_clase_datos.py
Nombre y dirección = Pepito Pérez Carrera 2 # 3 - 42
Edad = 25
Ciudad = Neira
Nombre y dirección = María Martínez Aquí va la dirección
Ciudad = Manizales
El nombre P1 es Pepito Pérez
La dirección de P2 es Aquí va la dirección
daalvarez@eredron:~ >
```

Agregando métodos a una clase

```
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:~ >
```



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.
- 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 `this`), ya que dentro de una clase estoy hablando de mi nombre, mi dirección, mi peso, y fuera de la clase estoy hablando del pero, nombre y dirección 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__()`

Constructores con parámetros

```
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)
```

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
```

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         return cadena
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.
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  p = Persona('Pepito Pérez', 20)
7  print(p)
```

Line: 8 of 11 Col: 1 LINE INS

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

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 instancia, 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.

HERENCIA

Por ejemplo, herencia de la clase C a la clase D, es la facilidad mediante la cual la clase D hereda en ella cada uno de los atributos y operaciones de C, como si esos atributos y operaciones hubiesen sido definidos por la misma D. Por lo tanto, puede usar los mismos métodos y variables registrados como "públicos" (*public*) en C. Los componentes registrados como "privados" (*private*) también se heredan pero se mantienen escondidos al programador y sólo pueden ser accedidos a través de otros métodos públicos. Para poder acceder a un atributo u operación de una clase en cualquiera de sus subclases pero mantenerla oculta para otras clases es necesario registrar los componentes como "protegidos" (*protected*), de esta manera serán visibles en C y en D pero no en otras clases.

```
 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.

Encapsulación de datos

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

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 accesar 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!