Herramientas de programación en Python

4. Introducción a programación orientada a objetos (OOP) clases y objetos en Python

Pedro Gomis

pedro.gomis@campusviu.es

Introducción a OOP, clases y objetos en Python

Aunque Python es, sin duda, un **lenguaje orientado a objetos (OO)**, hasta ahora se ha evitado intencionalmente el tratamiento de la **programación orientada a objetos** (OOP) en la enseñanza de programación con Python.

- A pesar de que evitamos la OOP en el tema 1 de Fundamentos de Programación, siempre ha estado presente en los ejercicios y tipos de datos usados.
- Utilizamos objetos y métodos de las clases sin explicar adecuadamente sus antecedentes de OOP.
- En esta sección nos pondremos al día con lo que faltaba hasta ahora.
- Se hará una introducción a los principios de la **programación orientada a objetos** en general y a los detalles del enfoque OOP de Python.
- OOP es una de las herramientas más poderosas de Python, sin embargo, no hace falta usarla, es decir, también se pueden escribir programas potentes y eficientes sin ella.

Introducción a OOP, clases y objetos en Python

Programación orientada a objetos (OOP)

- La Programación Orientada a Objetos (OOP) es un paradigma de programación en el que ***los conceptos del mundo real relevantes para nuestro problema se modelan a través de clases y objetos***, y en programas que incluyen una serie ***de interacciones entre estos objetos***.
- A finales de los años 70 y comienzos de los 80 el paradigma de la orientación a objetos se popularizó masivamente, primero con el lenguaje **Smalltalk** (Alan Kay en el mítico *laboratorio de Xerox*, Palo Alto), después con el **C++** (Bjarne Stroustrup en los también míticos *laboratorios Bell* de AT&T, 1983) y, por último, con el **Java**, uno de los lenguajes más populares actualmente (creado por James Gosling en *Sun Microsystems* en 1995).
- El uso de la orientación a objetos se popularizó también durante las tareas de **análisis de sistemas**, especialmente, a partir de la aparición en 1997 del lenguaje unificado de modelización (UML), creado por James Rumbaugh, Grady Booch e Ivar Jacobs.

Introducción a OOP, clases y objetos en Python

Programación orientada a objetos (OOP)

Clasificación

- Al utilizar orientación a objetos para **describir los objetos** que forman un **sistema** nos damos cuenta de que hay objetos que son de un **mismo tipo** y tienen una **misma estructura y responsabilidades**, es decir, atributos y funciones.
- La **clasificación** es el mecanismo por el que simplificamos la descripción de los objetos del sistema e identificamos aquellas clases de objetos que comparten las mismas responsabilidades (atributos y funciones o métodos).
- Por ejemplo, en un club de futbol una **clase** puede ser la **plantilla** de jugadores (**objetos**) y otra **clase** sea las diferentes **categorías**: juvenil, segunda plantilla y primera plantilla (**objetos**).
- Se denomina **clase** al grupo, e **instancia** al objeto.

Introducción a OOP, clases y objetos en Python

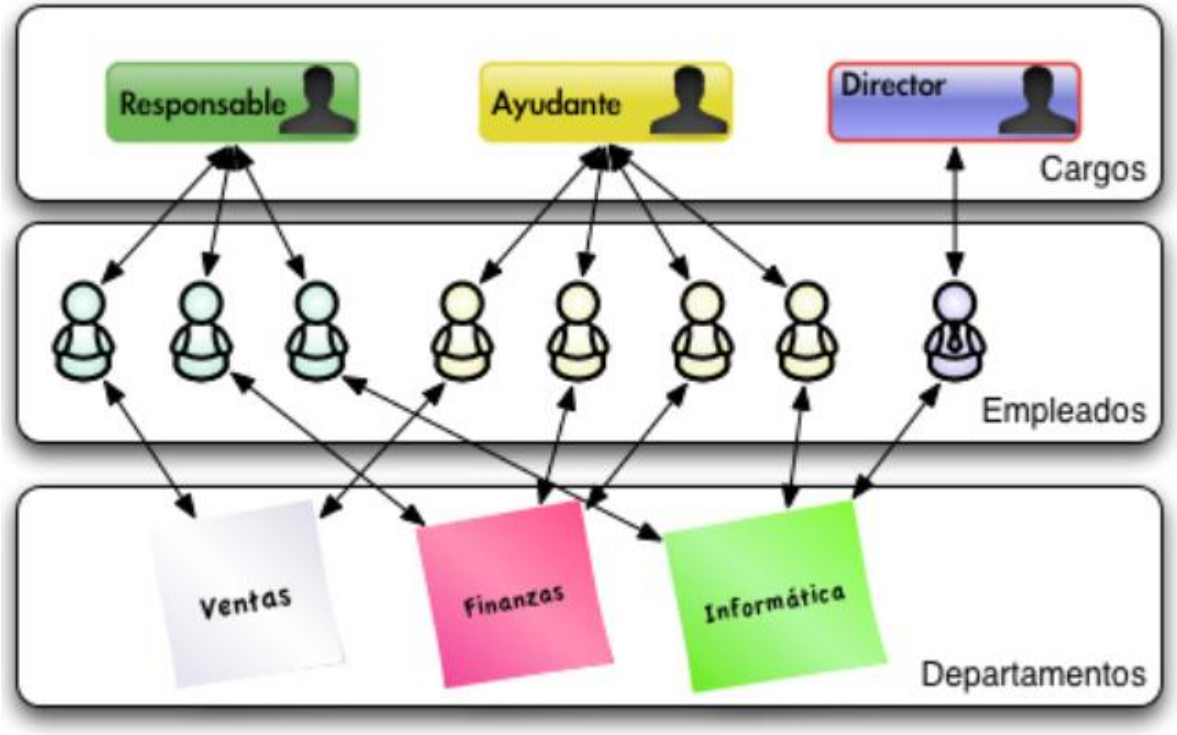
Programación orientada a objetos (OOP)

Clasificación - Ejemplo de una empresa*

Se tienen tres **clases** de **objetos**:

- Cargos
- Empleados
- Departamentos

De la clase Cargos hay tres instancias (Director, Ayudante y Responsable).
De Empleado, ocho insrancias, y de Departamento, tres instancias (Ventas, Finanzas e Informática)



* http://cv.uoc.edu/annotation/68cdc4848e7fd4149910ab359b7a710d/699675/PID_00232007/PID_00232007.html

Introducción a OOP, clases y objetos en Python

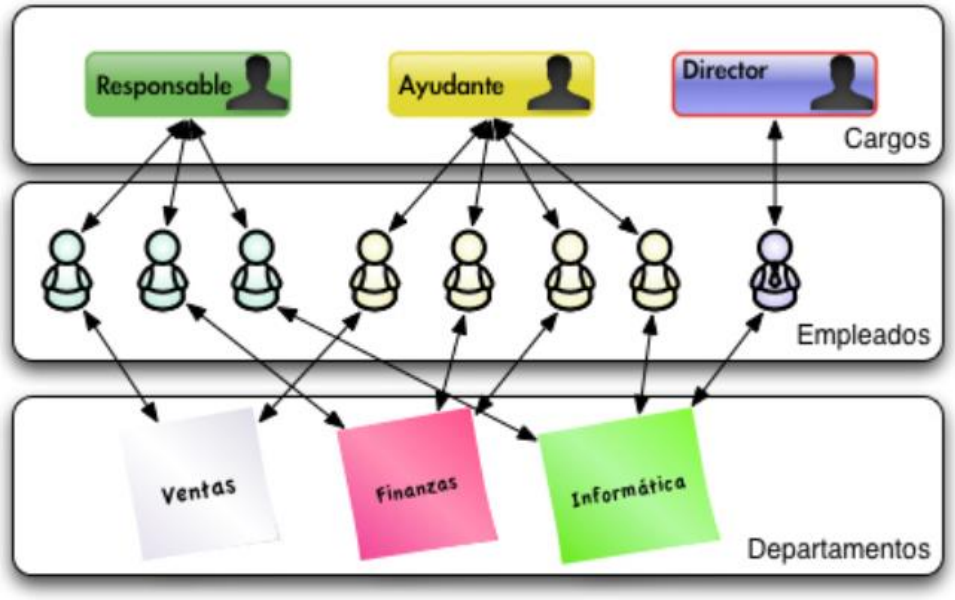
Programación orientada a objetos (OOP)

Clasificación - Ejemplo de una empresa*

Las responsabilidades de los **objetos** de una clase, las podemos dividir en tres tipos:

- La información que conocen (los datos, atributos o propiedades)
- Las tareas que pueden llevar a cabo (las operaciones, métodos o funciones)
- Los objetos que conocen (las asociaciones)

Un **Empleado** sabe cuál es su información (nombre, teléfono, etc.), existe una serie de tareas que puede llevar a cabo (redactar una solicitud, aprobarla, etc.) y está asociado con otras instancias (su cargo y su departamento)



* http://cv.uoc.edu/annotation/68cdc4848e7fd4149910ab359b7a710d/699675/PID_00232007/PID_00232007.html

Introducción a OOP, clases y objetos en Python

Programación orientada a objetos (OOP)

Clases y objetos

Podemos resumir entonces:

Objeto: es una entidad que agrupa un estado o información (datos, atributos o propiedades) y unas funcionalidades. *En Python, el estado del objeto se define a través de variables llamadas atributos, mientras que la funcionalidad se modela a través de funciones a las que se les conoce con el nombre de métodos del objeto.*

Clase: una plantilla genérica a partir de la cuál *instanciar* los **objetos**; plantilla que es la que define qué atributos y métodos tendrán los objetos de esa clase.

Otra definición de **clase** puede ser: "Descriptor de un conjunto de objetos que comparten los mismos atributos, métodos, relaciones y comportamiento".

Ejemplo: Existen un conjunto de distintos objetos que llamamos **coches** tipo turismo que tienen un conjunto de atributos comunes (marca, cilindrada, etc.) y funcionalidades comunes y a este conjunto lo llamamos **clase** coche.

Otros conceptos de la OOP: encapsulamiento, herencia y polimorfismo.

Introducción a OOP, clases y objetos en Python

OOP en Python

Conceptos previos: Espacios de nombres (*namespace*) y Ámbitos (*Scope*)

Las definiciones de clases en Python utilizan algunos trucos con los espacios de nombres, que son interesantes para saber cómo funcionan los alcances (*scope*) y espacios de nombres (*namespace*) en las clases.

Namespace: Un espacio de nombres (también llamado contexto) es un sistema de nombres para hacer que los nombres sean únicos para evitar ambigüedades. Es una relación de nombres a objetos.

Por ejemplo, la estructura de **directorios** de los sistemas de **archivos**. Se puede usar el ***mismo nombre*** de ***archivo*** en ***diferentes directorios***, a los archivos se puede acceder de manera única a través de los nombres de ruta (*pathname*).

Un **identificador** o **nombre** definido en un espacio de nombres está asociado con ese espacio de nombres. De esta manera, el mismo **identificador** puede definirse independientemente en múltiples espacios de nombres. (Como los mismos nombres de archivo en diferentes directorios).

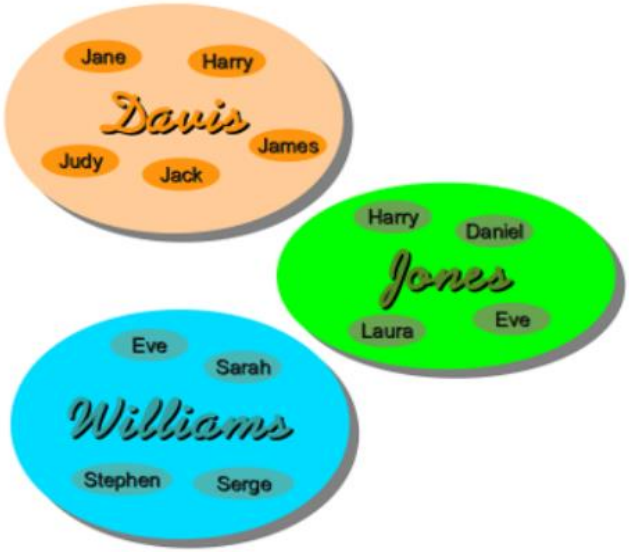
Introducción a OOP, clases y objetos en Python

OOP en Python

Conceptos previos: Espacios de nombres (*namespace*) y Ámbitos (*Scope*)

Namespace

Los espacios de nombres en Python se implementan como **diccionarios**: se definen mediante una asignación de nombres (las **claves** del diccionario) a los objetos (los **valores**).



Algunos espacios de nombres en Python:

- Nombres globales de un módulo
- Nombres locales en una invocación de función o método
- Nombres incorporados (*built-in*): este espacio de nombres contiene funciones built-in (por ejemplo, abs (), int (), max(), …)

Por *ejemplo*, dos módulos diferentes pueden tener ambos definidos una función `maximizar` sin confusión; los usuarios de los módulos deben usar el nombre del módulo como prefijo: `modulo1.maximizar`, `modulo2.maximizar`.

Introducción a OOP, clases y objetos en Python

OOP en Python

Conceptos previos: Espacios de nombres (*namespace*) y Ámbitos (*Scope*)

Namespace

Cuando se llama a una función, se crea un espacio de nombres local para esta función. Este espacio de nombres se borra si la función finaliza (ej, return).

Scope (Ámbito)

Un ámbito (*scope*) es una región de un programa donde se puede acceder directamente a un espacio de nombres, es decir, sin usar un prefijo de espacio de nombres.

En otras palabras: el ámbito de un nombre es el área de un programa donde este nombre puede usarse sin ambigüedades, por ejemplo, dentro de una función.

Introducción a OOP, clases y objetos en Python

OOP en Python Espacios de nombres (*namespace*) y Ámbitos (*Scope*)

Ámbito local

```
def myfunc():
    x = 300
    print(x)

myfunc()
```

```
300
```

```
def myfunc():
  x = 300
  def mi_func_inter():
    print(x)
  mi_func_inter()

myfunc()
```

```
300
```

Ámbito global

```
x = 300

def myfunc():
  x = 200
  print(x)

myfunc()

print(x)
```

```
200
300
```

Ámbito global

```
x = 300

def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

```
200
```

Introducción a OOP, clases y objetos en Python

OOP en Python Espacios de nombres (*namespace*) y Ámbitos (*Scope*)

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)
scope_test()
print("In global scope:", spam)
```

Ejemplo que muestra cómo hacer referencia a distintos ámbitos y espacios de nombres, y cómo las declaraciones **global** y **nonlocal** afectan la asignación de variables

Notar cómo la asignación local (default) no cambió la vinculación de spam de scope_test(). La asignación nonlocal cambió la vinculación de spam de scope_test(), y la asignación global cambió la vinculación a nivel de módulo.

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Introducción a OOP, clases y objetos en Python

OOP en Python. Siempre primera clase

En Python todo son **clases**. Guido van Rossum diseñó el lenguaje según el principio ***todo de primera clase***.

Guido escribió: ***Una de mis metas para Python era hacer que todos los objetos fueran de "primera clase"***. Con esto, quiso decir que quería todos los objetos que pudieran usarse en el lenguaje (por ejemplo, enteros, strings, funciones, clases, módulos, métodos, etc.) que puedan tener el mismo estado.

Es decir, "pueden asignarse a variables, colocarse en listas, almacenarse en diccionarios, pasarse como argumentos, y así sucesivamente". (Blog, *The History of Python*, 27 de febrero de 2009)*

Esto significa que **todo** se trata de la misma manera, todo es una **clase**: las funciones y los métodos son valores como listas, enteros o reales. Cada uno de estos son instancias de sus clases correspondientes.

* http://python-history.blogspot.com/2009/02/

Introducción a OOP, clases y objetos en Python

OOP en Python

Ejemplos de instancias (objetos) de diferentes clases:

```
>>> x = 25
>>> type(x)
<class 'int'>
>>> y = 2.5
>>> type(y)
<class 'float'>
>>> def f(x):
        return x+1

>>> type(f)
<class 'function'>
>>> import math
>>> type(math)
<class 'module'>
>>>
```

Introducción a OOP, clases y objetos en Python

OOP en Python

Una de las muchas **clases** integradas en Python es la clase list, que hemos utilizado en muchos ejercicios y ejemplos. La clase list proporciona una gran cantidad de métodos para crear listas, acceder y cambiar elementos, o eliminar elementos:

```
>>> x = [3, 6, 9]
>>> y = [45, "abc"]
>>> id(x)
2041851444808
>>> type(x)
<class 'list'>
>>> x.append(42)
>>> ultimo = y.pop()
>>> print(ultimo)
abc
```

- Las variables **x** e **y** del ejemplo denotan dos **instancias** de la **clase list**. En términos simplificados, hemos dicho hasta ahora que "x e y son listas".
- Usaremos los términos **objeto** e **instancia** como sinónimos.
- Los **objetos** son la abstracción de Python para los **datos**. Cada **objeto** tiene una **identidad**, un **tipo** y un **valor**.

Introducción a OOP, clases y objetos en Python

Primer vistazo a las clases

Sintaxis de definición

Las clases introducen algo de sintaxis nueva

```
class NombreClase:
    <sentencia-1>
    ...
    <sentencia-N>
```

En la práctica, las declaraciones dentro de una definición de clase generalmente serán definiciones de funciones, pero se permiten otras declaraciones, y a veces son útiles.

Las definiciones de funciones dentro de una clase normalmente tienen una lista de **argumentos** peculiar, dictada por las convenciones de invocación de métodos.

Cuando se define una **clase**, se crea un nuevo espacio de nombres (***namespace***), el cual se usa como ámbito (***scope***) local; por lo tanto, todas las asignaciones a variables locales van a este nuevo *namespace*. En particular, las definiciones de funciones asocian el nombre de las funciones nuevas en este *namespace*.

Introducción a OOP, clases y objetos en Python

Primer vistazo a las clases

Crear Clases

Las ***clases*** soportan dos tipos de operaciones: hacer **referencia a atributos** e **instanciación**
Para *hacer* ***referencia a atributos*** se usa la sintaxis estándar de todas las referencias a atributos en Python: `objeto.nombre`. Los nombres de atributo válidos son todos los nombres que estaban en el ***espacio de nombres*** de la clase cuando ésta se creó. Por lo tanto, si la definición de la clase es:

```
class MiClase:
    """Ejemplo simple de clase"""
    i = 12345
    def f(self):
        return 'hola mundo'
```

entonces `MiClase.i` y `MiClase.f` son **referencias de atributos** válidos, que devuelven un entero y un objeto función, respectivamente.

__doc__ también es un atributo válido que devuelve el *docstring* de la clase:

```
>>> MiClase.i
12345
>>> MiClase.__doc__
'Ejemplo simple de clase'
```

Introducción a OOP, clases y objetos en Python

Primer vistazo a las clases

Crear objetos

La ***instanciación*** de clases usa la **notación de funciones**. Pensarlo como que el objeto de clase es una función sin parámetros que devuelve una nueva **instancia** de la clase.

```
>>> x = MiClase()
>>> x.i
12345
>>> x.f()
'hola mundo'
```

La primera sentencia crea una nueva *instancia* de la clase y asigna este objeto a la variable local `x`.

Los atributos de clase también pueden ser asignados, o sea que se pueden cambiar el valor de `MiClase.i` mediante asignación.

```
>>> MiClase.i = 456
>>> x = MiClase()
>>> x.i
456
```

Introducción a OOP, clases y objetos en Python

Primer vistazo a las clases

Crear objetos

La operación de ***instanciación*** crea un **objeto vacío**. Muchas clases necesitan crear objetos en un **estado inicial** particular. Por lo tanto una clase puede definir un **método especial** llamado __init__(), de esta forma:

```
def __init__(self):
    self.datos = []
```

Lo insertamos en el `"""Ejemplo simple de clase"""`

```
class MiClase:
    """Ejemplo simple de clase"""
    i = 12345
    def __init__(self):
        self.datos = []
    def f(self):
        return 'hola mundo'
```

Cuando en una clase se define un **método** __init__(), la **instanciación** de la **clase** invoca automáticamente a __init__() para la instancia recién creada. En el ejemplo:

```
>>> x = MiClase()
>>> x.datos
[]
```

Introducción a OOP, clases y objetos en Python

Primer vistazo a las clases

Crear clases y objetos

El **método** __init__() puede también tener argumentos para mayor flexibilidad. En ese caso, los argumentos que se pasen al operador de instanciación de la clase van a parar al método __init__(). Por ejemplo:

```
class Complejo:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
```

```
>>> z = Complejo(3.0, -4.5)
>>> z.r, z.i
(3.0, -4.5)
```

El parámetro *self* es una referencia a la instancia actual de la clase y se usa para acceder a las variables que pertenecen a la clase. Esta es una idea inspirada en Modula-3 y sirve para referirse al objeto actual. Este mecanismo es necesario para poder acceder a los atributos y métodos del objeto diferenciando, por ejemplo, una variable local mi_var de un atributo del objeto *self*.mi_var.

Introducción a OOP, clases y objetos en Python

Primer vistazo a las clases

Atributos y métodos de objetos

Atributos

Los objetos soportan dos tipos de referencia de atributos: **atributos de datos** y **métodos**. Los atributos de datos de los objetos no necesitan ser declarados; tal como las variables locales son creados la primera vez que se les asigna algo. Por ejemplo, si x es la instancia de `MiClase` creada antes, se puede **crear un atributo** al objeto x de la forma:

```
>>> x.contador = 1
```

En el ejemplo de la clase Complejo hemos creado dos **atributos**, realpart e imagpart y los objetos que se puedan crear los tendrán. Pero se pueden agregar nuevos atributos:

```
>>> z = Complejo(3.0, -4.5)
>>> z.r, z.i
(3.0, -4.5)
>>> z.nombre = 'ComplejoZ’
>>> print(z.nombre)
ComplejoZ
```

Introducción a OOP, clases y objetos en Python

Primer vistazo a las clases

Atributos y métodos de objetos

Métodos

El otro tipo de referencia de atributo de instancia es un **método**. Un método es una función que "pertenece" a un objeto. En el ejemplo `MiClase`, el método `f()`:

```
   ...
  def f(self):
       return 'hola mundo'
```

```
>>> x.f()
'hola mundo'
```

Pero no es necesario llamar al método justo en ese momento: en el siguiente ejemplo `x.f` es un *objeto método*, y puede ser guardado y ser llamado más tarde.

```
>>> xf = x.f
>>> print(xf())
hola mundo
```

Hemos visto que un **método** difiere de una **función** solo en dos aspectos:

- Pertenece a una clase y se define dentro de una clase.
- El primer parámetro en la definición de un método tiene que ser una referencia a la instancia, que llamó al método. Este parámetro generalmente se llama "*self*".

Introducción a OOP, clases y objetos en Python

Métodos: De hecho, "***self***" no es una palabra clave de Python. ¡***Es solo un nombre por convención***! Por lo tanto, los programadores de C ++ o Java son libres de llamarlo "this", pero de esta manera se arriesgan a que otros tengan mayores dificultades para comprender su código.

Modificar atributos de objetos

Se pueden modificar atributos de los objetos. Por ejemplo en el objeto **z** de la clase Complejo:

```
>>> z.r = 7.5
>>> print(z.r)
7.5
```

Eliminar Objetos y propiedades de los objetos

Se pueden eliminar propiedades en objetos utilizando la palabra clave `del`:

```
>>> del x.contador
```

También se pueden eliminar objetos.

```
>>> del x
```

Introducción a OOP, clases y objetos en Python

Primer vistazo a las clases

Variables de clases y de instancias

Las variables de instancia son datos únicos de cada instancia y las variables de las clases son para atributos y métodos compartidos por todas las instancias de la clase:

```
class Perro:

    tipo = 'canino'    # variable de clase que comparten las instancias

    def __init__(self, nombre):
        self.nombre = nombre # variables en instancias únicas de c/u

>>> d = Perro('Roc')
>>> e = Perro('Luna')
>>> d.tipo     # compartido por todos los perros
'canino'
>>> d.nombre   # único de d
'Roc'
>>> e.nombre   # único de e
'Luna'
```

Introducción a OOP, clases y objetos en Python

Primer vistazo a las clases - Variables de clases y de instancias

Si hay objetos ***mutables*** (como listas) puede haber **problemas** cuando se **comparte** una **variable** de la clase con todas las instancias. En el ejemplo de la clase perro, si en lugar de tipo (string) tuviéramos la variable trucos (lista):

```
class Perro:

    trucos = []   # uso erróneo de variable en una clase

    def __init__(self, nombre):
        self.nombre = nombre    # variables en instancias únicas de c/u

    def agrega_truco(self, truco):  # método
        self.trucos.append(truco)

>>> d = Perro('Roc')
>>> e = Perro('Luna')
>>> d.agrega_truco('da vueltas')
>>> e.agrega_truco('salta la cuerda')
>>> d.trucos   # resultado no deseado, compartido por todos los perros!
['da vueltas', 'salta la cuerda']
```

Introducción a OOP, clases y objetos en Python

Primer vistazo a las clases - Variables de clases y de instancias

El diseño **correcto** de la clase debería usar una variable de instancia en lugar de clase:

```
class Perro:
    """ Ejemplo de diseño de clase: perro"""

    def __init__(self, nombre):
        self.nombre = nombre
        self.trucos = []

    def agrega_truco(self, truco):
        self.trucos.append(truco)

>>> d = Perro('Roc')
>>> e = Perro('Luna')
>>> d.agrega_truco('da vueltas')
>>> e.agrega_truco('salta la cuerda ')
>>> d.trucos
['da vueltas']
>>> e.trucos
['salta la cuerda']
```

Introducción a OOP, clases y objetos en Python

Ejemplo

```
Class Persona:
    def __init__ (self, nombre, dni, edad):  #Atributos: nombre,dni,edad
        self.nombre = nombre
        self.dni = dni
        self.edad = edad
    def iniciales(self):                # Métodos
        txt = ''
        for caracter in self.nombre:
            if caracter >= 'A' and caracter <= 'Z':
                txt += caracter +'. '
        return txt
    def iniciales2(self):
        lst = self.nombre.split()
        txt = lst[0][0]+'.'+ lst[1][0]+'.'
        return txt
    def esMayorEdad(self):
        return self.edad >= 18

#Objetos
Iniesta = Persona('Andrés Iniesta', '1234S', 35)
Sergio = Persona('Sergio Ramos', '5678C', 34 )
Saul = Persona('Saúl Ñiguez','7891W', 25)
```

Introducción a OOP, clases y objetos en Python

Ocultación de información y encapsulamiento

Abstracción

En orientación a objetos, la **abstracción** es el mecanismo por el que decidimos qué responsabilidades asociadas a una clase formarán parte de su definición.

Ocultación de información

Cada elemento (módulo o subsistema) del sistema contiene una cierta información que "oculta" al resto de los módulos y les presenta una visión simplificada.

Encapsulamiento

La encapsulación se refiere a impedir el acceso a determinados métodos y atributos de los objetos, estableciendo así, qué puede utilizarse desde fuera de la clase.

- La encapsulación se ve como la agrupación de datos con los métodos que operan en esos datos.
- El ocultamiento de información es el principio de que cierta información interna o datos están "ocultos", por lo que no se puede cambiar accidentalmente.
- La abstracción de datos está presente, si se utilizan tanto la ocultación de datos como la encapsulación de datos. Esto significa que la abstracción de datos es el término más amplio:
  Abstracción de datos = Encapsulamiento + Ocultamiento de información

Introducción a OOP, clases y objetos en Python

Ocultación de información y encapsulamiento

Encapsulamiento

El **encapsulamiento** nos permite ocultar información en otros objetos definiendo qué atributos, operaciones y asociaciones de un objeto les serán visibles y cuáles estarán ocultos.

Se denomina **visibilidad** de un atributo o método a la propiedad que define qué objetos pueden verlo.

Los diferentes lenguajes de programación definen diferentes tipos de **visibilidad**, pero los casos más típicos son:

- Visibilidad **pública**: cualquier objeto tiene acceso.
- Visibilidad **privada**: sólo los objetos de la misma clase tienen acceso.
- Visibilidad **protegida**: sólo los objetos de la misma clase o de alguna subclase tienen acceso.

En Java (UML):

- Private (-): solo deben ser utilizados por el propietario (dentro de la definición de clase)
- Public (+) : los atributos públicos pueden y deben usarse libremente
- Protected (#): pueden usarse pero bajo ciertas condiciones

Introducción a OOP, clases y objetos en Python

Ocultación de información y encapsulamiento

Encapsulamiento

Python usa un esquema de nombres especial para los atributos con el fin de controlar la accesibilidad de los mismos.

Hasta ahora, hemos usado nombres de atributos, que se pueden usar libremente dentro o fuera de una definición de clase, como hemos visto. Esto corresponde a los atributos **públicos**, por supuesto.

Hay dos formas de restringir el acceso a los atributos de clase:

- Primero, podemos prefijar un nombre de atributo con un guión bajo "_". Esto marca el atributo como **protegido**. Le dice a los usuarios de la clase que no usen este atributo a menos que alguien escriba una subclase (herencia y subclases luego).
- En segundo lugar, podemos prefijar un nombre de atributo con dos guiones bajos "__". El atributo ahora es **inaccesible e invisible** desde el exterior. No es posible leer ni escribir en esos atributos, excepto dentro de la propia definición de clase

Identificador	Tipo	Significa
nombre	público	Se puede usar libremente desde fuera o dentro de la clase
_nombre	protegido	No debe usarse fuera de la clase
__nombre	privado	Atributo inaccesible e invisible

Introducción a OOP, clases y objetos en Python

Encapsulamiento. Ejemplo en Python

Creamos el siguiente código y los guardamos como prueba_atributos.py

```
class A():
    def __init__(self):
        self.__priv = "Soy privado"
        self._prot = "Estoy protegido"
        self.pub = "Soy público"
```

```
>>> from prueba_atributos import A
>>> x = A()
>>> x.pub
'Soy público'
>>> x.pub = x.pub + " y mi valor se puede cambiar"
>>> x.pub
'Soy públco y mi valor se puede cambiar'
>>> x._prot
'Estoy protegido'
>>> x.__priv
Traceback (most recent call last):
  File "<ipython-input-69-f75b36b98afa>", line 1, in <module>
    x.__priv
AttributeError: 'A' object has no attribute '__priv'
```

Introducción a OOP, clases y objetos en Python

Herencia

En un lenguaje orientado a objetos cuando hacemos que una clase (**subclase**) herede la definición de otra clase (**superclase**), de manera que se aplique a instancias de éstas, estamos haciendo que la subclase **contenga todos los atributos y métodos** que tenía la superclase. No obstante al acto de heredar de una clase también se le llama a menudo "extender o **derivar una clase**".

Este mecanismo nos permite, a la hora de definir una subclase, indicar sólo aquellas características que son específicas, mientras que las que son comunes con otras subclases de la misma superclase, decimos que las **hereda**.

Se llama también **clase padre a la superclase** y **clase hijo a la subclase**.

En Python la sintaxis es:

```
class ClaseDerivada(ClaseBase):
    ...
```

Ejemplo:

```
class MiClaseHija(Miclase):
    pass
```

Si queremos heredar los atributos y métodos y agregar otros nuevos se puede usar la función `super()` como se verá en otro ejemplo.

Introducción a OOP, clases y objetos en Python

Herencia Ejemplos

```
class Instrumento:
    def __init__ (self, clase, nombre, precio):
        self.clase = clase
        self.nombre = nombre
        self.precio = precio
    def tocar(self):
        print("Tocando música")
    def romper(self):
        print("¡Que has hecho! los", self.precio,"€ los vas a pagar")
"""
Heredar o extender/derivar una clase
"""
class bateria(Instrumento):
    pass                    # Hereda atributos
class guitarra(Instrumento):
    # Si está solo esto, sobreescribe los atributes de la clase principal
    def __init__ (self, clase, nombre, precio, tipo):
        super().__init__(clase, nombre, precio)   # Hereda atributos
        self.tipo = tipo
```

Introducción a OOP, clases y objetos en Python

Polimorfismo

El **polimorfismo** es la capacidad de un objeto de presentarse con formas diferentes (interfaces diferentes) según el contexto.

Las funciones o métodos polimórficos se pueden aplicar a argumentos de diferentes tipos, y pueden comportarse de manera diferente dependiendo del **tipo** de argumentos a los que se aplican. También podemos definir el mismo nombre de función con un número variable de parámetros.

En Python, por ejemplo:

```
def f(x, y):
    print("valores: ", x, y)

f(42, 43)
f(42, 43.7)
f(42.3, 43)
f([3,5,6],(3,5))
```

Podemos llamar a esta función con varios **tipos de datos**, como se demuestra en el ejemplo. En lenguajes de programación ***tipados*** como Java o C ++, tendríamos que sobrecargar (*overloading*) **f()** para implementar las diversas combinaciones de tipos.

Python es implícitamente **polimórfico**

Introducción a OOP, clases y objetos en Python

Misceláneos

Record y *structure*

A veces es útil tener un tipo de datos similar al **registro** (**record**) de **Pascal** o la **estructura** (**structure**) de **C** o **Matlab**, que sirva para juntar algunos ítems con nombre. Una definición de clase vacía funcionará perfectamente en este caso.

```
class Empleado:
    pass

juan = Empleado() # Crear un registro de empleado vacío
# Llenar los campos del registro
juan.nombre = 'Juan Martínez'
juan.depto = 'Sistemas de control'
juan.salario = 1500
```

Comentarios finales

- Además de **programar**, hay **que diseñar el software**.
- El **diseño** significa **estructurar el código** de tal manera que sea **fácil** de **entender**, fácil de **cambiar** para cumplir con los nuevos **requisitos**, fácil de **probar**, etc.
- La **programación orientada a objetos** (OOP) es un **paradigma** de **programación** en el que los **conceptos** del mundo real relevantes para nuestro problema se **modelan** a través de **clases** y **objetos**, y en programas que incluyen una serie de interacciones entre estos objetos.
- **OOP** es una de las **herramientas** más **poderosas** de **Python**, sin embargo, ***no hace falta usarla***, es decir, también ***se pueden escribir programas potentes y eficientes sin ella***.