

Funciones en Python

[9.1] ¿Cómo estudiar este tema?

[9.2] ¿Qué es la programación orientada a objetos?

[9.3] Conceptos fundamentales

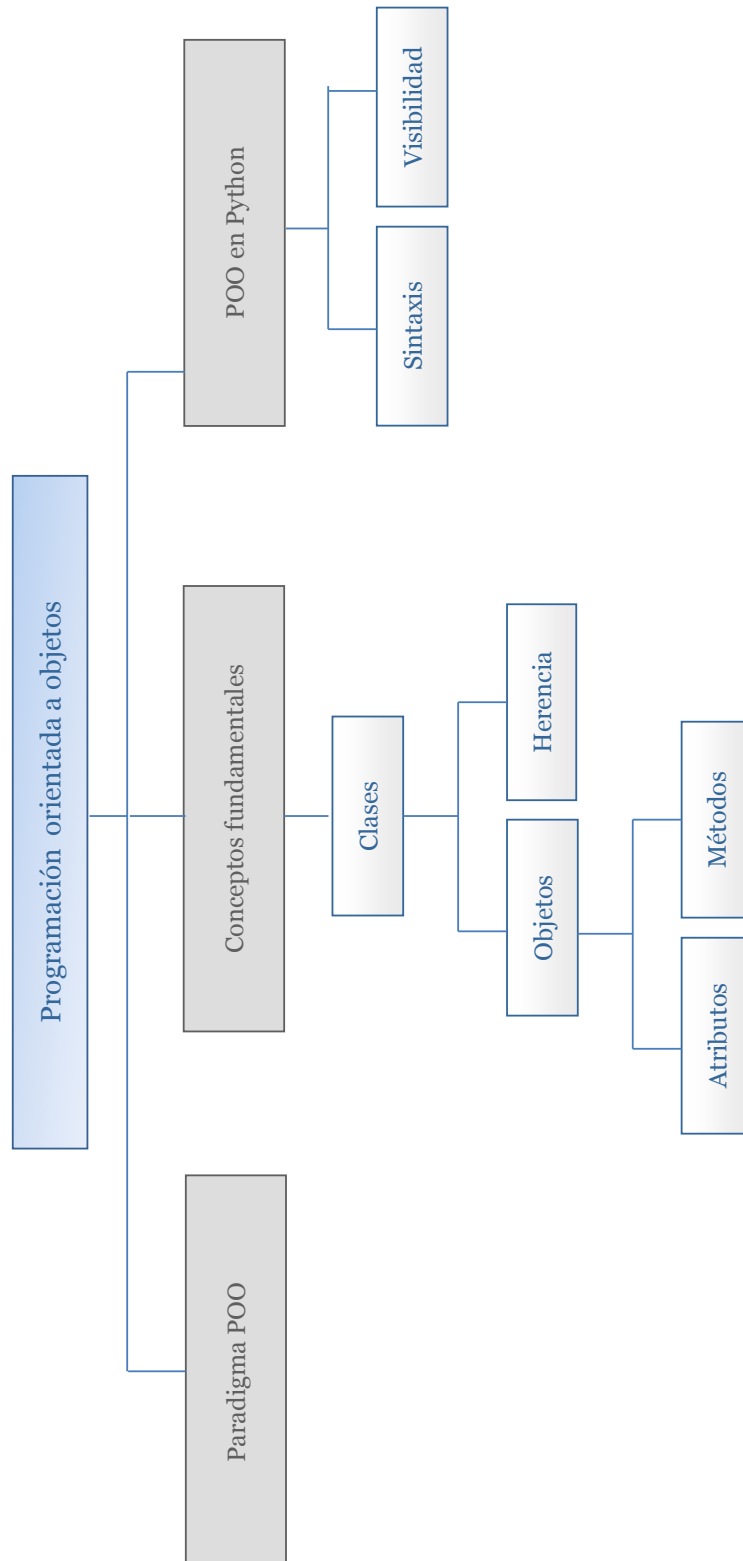
[9.4] POO en Python

[9.5] Referencias bibliográficas

9

TEMA

Esquema



Ideas clave

9.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás comprender las **Ideas clave** expuestas en este documento, que se complementan con lecturas y otros recursos para que puedas ampliar los conocimientos sobre el mismo.

En este tema se estudia el paradigma de la **programación orientada a objetos**, sus principios, los conceptos básicos y cómo implementarlo en Python. Por tanto, para comprender este tema, es esencial comprender.

- » En qué consiste el paradigma de la **programación orientada a objetos**.
- » Conceptos fundamentales: **clases, objetos, atributos, métodos, herencia**.
- » Cómo se implementa lo aprendido en Python: **sintaxis y visibilidad** de objetos y métodos.

Para una comprensión óptima del tema, se recomienda ir ejecutando las clases y comandos que se muestran como ejemplo.

9.2. ¿Qué es la programación orientada a objetos?

Como ya hemos visto, la **programación estructurada** está centrada en el desarrollo de **algoritmos**. A estos algoritmos se les pasan datos de entrada que, más allá de verificar que tienen un formato adecuado, no tienen por qué cumplir nada más, es decir no tienen por qué devolver una salida que tenga sentido. Por ejemplo, si se tienen los datos relativos al peso, la estatura y la edad de una persona, se pueden pasar estos tres datos como entrada para calcular la media aritmética y el algoritmo funcionará perfectamente a pesar de que el cálculo no tenga utilidad.

Otra desventaja de la programación estructurada es que las **funciones tienen acceso ilimitado a los datos globales**, que pueden ser modificados accidentalmente, comprometiendo al resto del programa.

En resumen, por una parte se tienen los datos y por otra los algoritmos que los manipulan (que también son inconexos entre sí). Esto proporciona un **modelado** bastante **pobre de la realidad**, ya que en realidad las modificaciones que se realizan sobre un tipo de datos están estrechamente ligadas al dato en sí.

En la programación orientada a objetos (POO), que es un paradigma de programación que surge en la década de los 70, estos problemas se evitan.

El paradigma de POO se centra en los datos y se intenta ajustar el lenguaje a cada problema. Esto se hace diseñando formatos de datos que representen la realidad que se quiere modelar.

Por ejemplo, si se quiere hacer un programa que gestiona los pacientes de un hospital, el foco no se pone en las acciones que se quiere que lleve a cabo el programa, sino que se pone en el objeto, que se corresponde con la parte de la realidad que se quiere modelar (pacientes, en este caso). Así, cada paciente será un objeto con unas características (atributos) determinadas: edad, peso, grupo sanguíneo, etc. sobre el que se pueden realizar algunas tareas específicas (métodos) como dar el alta, dar la baja, citar para consulta, etc.

Nótese que no se van a definir los métodos (como por ejemplo, dar de alta) por separado, sino que están ligados al objeto paciente, de forma que no se podrán aplicar a otro objeto del hospital, como puede ser el laboratorio. Es en este sentido en el que se dice que la POO modela mejor la realidad, porque en el mundo real no separamos los objetos de las acciones que podemos realizar sobre ellos: no tendría sentido citar para consulta al laboratorio de un hospital y esto se refleja en la programación orientada a objetos.

Además, se debe evitar que cualquier otro método trabaje sobre los pacientes. Tal y como se haría en la gestión manual de los pacientes, no se permitiría que estos datos fueran modificados, por ejemplo, por métodos pertenecientes a la gestión del servicio de comidas.

Una vez introducidos los conceptos generales de forma intuitiva, veamos sus definiciones más en detalle.

9.3. Conceptos fundamentales

Objetos

Hemos visto que en el paradigma POO el foco está en el objeto, pero, ¿qué es exactamente un objeto? Intuitivamente, podemos decir que cualquier concepto puede ser un objeto. Por tanto, objetos pueden ser:

- » Personas: pacientes, estudiantes, ciudadanos, menores de edad, presos, etc.
- » Animales: mamíferos, fauna africana, aves, mascotas, etc.
- » Objetos tangibles: mobiliario, vehículos, libros, cuadros, etc.
- » Objetos espaciales: puntos en el espacio, mapas, altitudes, etc.
- » Interfaces gráficos: menús, cuadros de diálogo, ventanas, etc.
- » Lugares: muelle de carga, zona de picking, puerta de embarque, etc.
- » Organizaciones: empresa, departamento, grupo de trabajo, etc.
- » Papeles representados por personas: cliente, cliente interno, empleado, jefe, autónomo, etc.

Según una definición más formal, un **objeto** combina, en una sola unidad, **los datos y las funciones que operan sobre esos datos**.

Los **datos** pueden ser de cualquiera de los tipos ya estudiados: caracteres, enteros, listas, etc. y también pueden contener otros objetos.

Los **atributos** sirven para describir el estado de un objeto, es decir, son datos o variables que sirven para caracterizar al objeto.

En el ejemplo de los pacientes, los atributos serían Edad, Peso, Grupo sanguíneo, etc.

Las **funciones**, también llamadas métodos, son las que realizan acciones (lectura, modificación, escritura) sobre los datos.

Volviendo al ejemplo del hospital las funciones serían dar de baja o de alta, solicitar una cita, etc. (Figura 7.1).

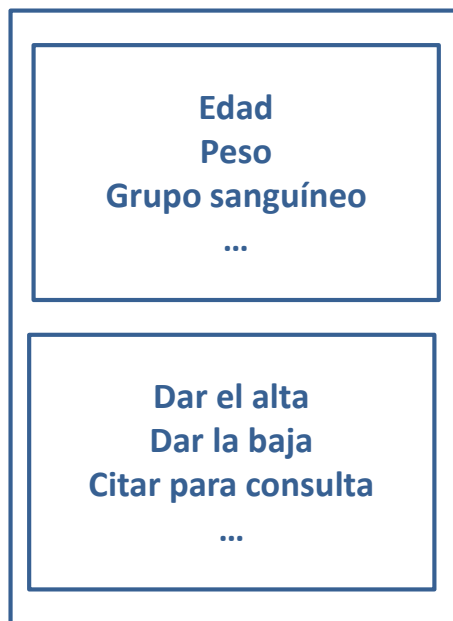


Figura 7.1: Representación de algunos atributos y funciones del objeto paciente

En algunos lenguajes de programación, como C++ o Java, los datos y funciones de un objeto pueden ser públicos o privados. Solo las funciones del objeto pueden realizar modificaciones sobre datos privados, garantizando así que no puedan ser modificados accidentalmente por otras partes del programa. Esto se conoce con el nombre de **encapsulamiento** (Figura 7.2). Aunque en Python no es posible realizar esta distinción, hay que tratar de evitar igualmente la manipulación accidental de los datos.

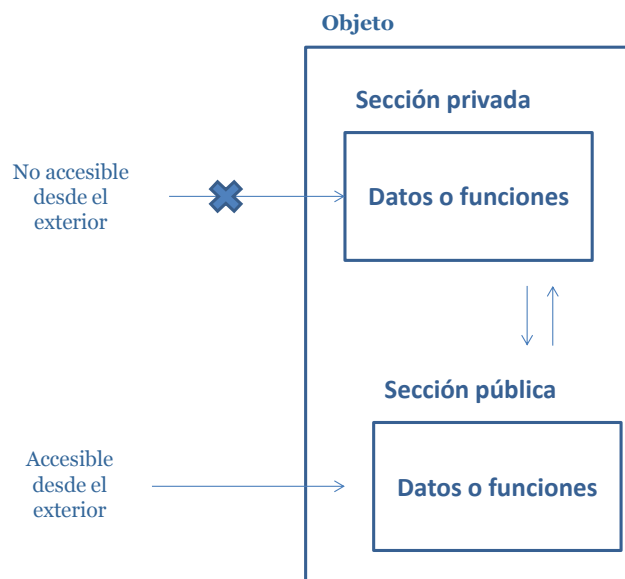


Figura 7.2: Representación de las secciones pública y privada de un objeto

En resumen, un objeto es la modelización de un determinado concepto y contiene toda la información relevante para que el programa pueda trabajar con él como se desea. Esta información está compuesta por los datos que forman sus atributos y la funciones, que describen qué operaciones pueden realizarse con los objetos. Además, con el encapsulamiento, puede evitarse que ciertas partes del objeto puedan ser utilizadas fuera de él.

Métodos y mensajes

Para ilustrar los métodos y mensajes en la POO, vamos a considerar el ejemplo de un cliente en un restaurante, viendo esta situación en clave de objetos.

El objeto cliente explica al objeto camarero qué platos de la carta quiere comer. El cliente no tiene que decirle al camarero cómo anotar la comanda ni qué debe hacer a continuación, es tarea del camarero resolver el problema. El cliente simplemente ha emitido un mensaje.

El camarero va a la cocina y comunica al cocinero qué platos deben serle entregados. En este punto, el camarero a su vez no dice al cocinero cómo tiene que sazonar la comida o cuándo sacar el plato del horno, ya que es el cocinero el encargado de proporcionar la comida. Es decir, el objeto camarero ha enviado un mensaje al objeto cocinero, que debe encargarse por su cuenta de resolver el problema.

En este ejemplo hay tres objetos distintos (cliente, camarero y cocinero) que tienen definidos métodos distintos (un cliente no puede entrar en la cocina y preparar su propia comida, por ejemplo). Además, para realizar una acción en concreto, cada objeto solo tiene que enviar un mensaje a otro objeto, no necesita saber cómo tiene que hacerse todo el proceso.

Como puede deducirse del ejemplo anterior, un mensaje es la forma que tienen los objetos de un programa de interactuar entre sí.

Un **mensaje** representa la **acción que debe realizar un objeto** a petición de otro.

Los **métodos** se definen como **el conjunto de funciones** (o de procedimientos y funciones, según el lenguaje) que son propios **de un objeto** y que determinan las acciones que deben llevarse a cabo cuando se reciba un mensaje.

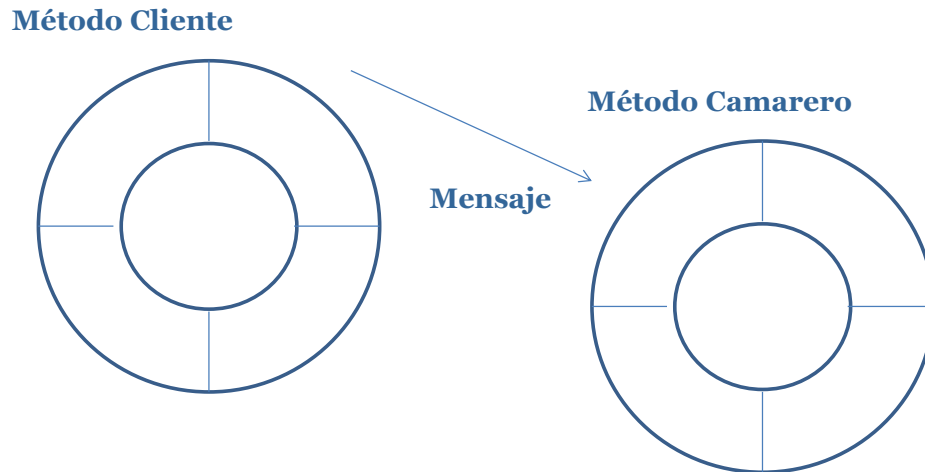


Figura 7.3: Diagrama de mensajes entre objetos

El hecho de que un objeto no sepa qué operaciones realiza otro objeto para devolverle lo que le ha pedido, hace que la estructura interna de los objetos funcione como una caja negra para usuarios y programadores externos.

Clases

«Una **clase** es la descripción de un conjunto de objetos: consta de métodos y datos que resumen características comunes de un conjunto de objetos. Se pueden definir muchos objetos de la misma clase. Dicho de otro modo, una clase es una declaración de un tipo objeto. (Aguilar, 2008).»

En el ejemplo de los pacientes del hospital, cada paciente sería un objeto y la clase sería el conjunto de atributos (Edad, Peso, Grupo sanguíneo, etc.) de cada objeto y de métodos (dar el alta o la baja, citar para consulta, etc.) que operan sobre los objetos.

Es decir, Juan Fernández Soler, de 33 años, 78kg y grupo sanguíneo A+ es un objeto de la clase paciente, como también lo es Pepa Marcos, de 45 años, 50kg y grupo sanguíneo A+.

La clase puede verse como la **plantilla** a partir de la cual se construyen los objetos de dicha clase (Figuras 7.4 y 7.5). Cada vez que se define una variable a partir de una clase se dice que se está creando una instancia de dicha clase. Por tanto, **los objetos son instancias de las clases**. Los valores que toman los atributos en un determinado objeto determinan su **estado**. Un atributo consta de dos partes: nombre de atributo y valor de atributo.

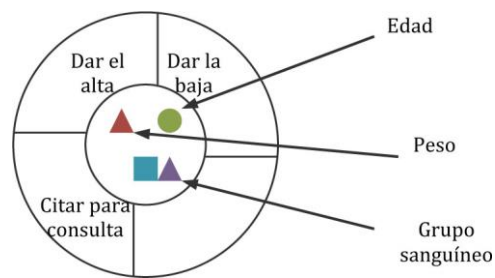
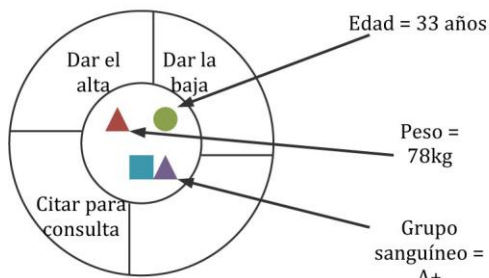


Figura 7.4: Definición de la clase Paciente

Juan Fernández Soler



Pepa Marcos

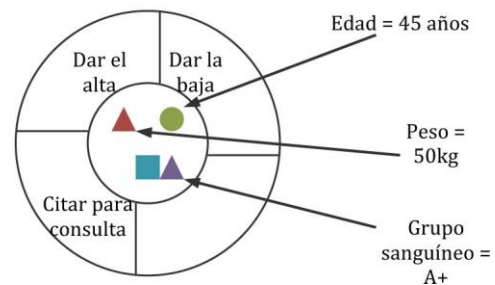


Figura 7.5: Objetos o instancias de la clase Paciente

Herencia

Con la **herencia** es posible definir nuevas clases a partir de clases ya existentes.

Cuando se crea una clase a partir de otra, la clase nueva hereda las características de la clase base, es decir los atributos y métodos que tuviera. Esto es muy útil porque permite reutilizar el código ya implementado.

Volviendo al ejemplo del hospital, se pueden clasificar los pacientes en función de la especialidad que requieran: cardiología, traumatología, ginecología, etc. Por tanto, de la superclase **Paciente** se pueden definir tantas subclases como se necesite. Todos los atributos y métodos de la clase **Paciente** se heredarán, pero además las nuevas clases pueden incorporar nuevos métodos (como dar un volante para realizar un electrocardiograma) y nuevos atributos (paciente fumador o no fumador) (Figura 7.6).

Por tanto, cuando se habla de herencia, hay que tener en cuenta la jerarquía de clases (compuesta por la clase base y las derivadas). Además del código y los datos que se heredan, también se pueden cambiar la funcionalidad de los métodos de la clase base que se considere conveniente. Esto recibe el nombre de sobrescritura.

Por ejemplo, supongamos que todas las citas para consulta se conciertan igual en todas las especialidades menos en cardiología, que hay que llamar a un número de teléfono diferente. En este caso es posible sobrescribir el método heredado `Citar para consulta` en la clase `Pacientes cardiología`. Es decir, simplemente se trata de redefinir el método, pero manteniendo el nombre, lo que permite mayor claridad en el código y un mantenimiento del código más sencillo.

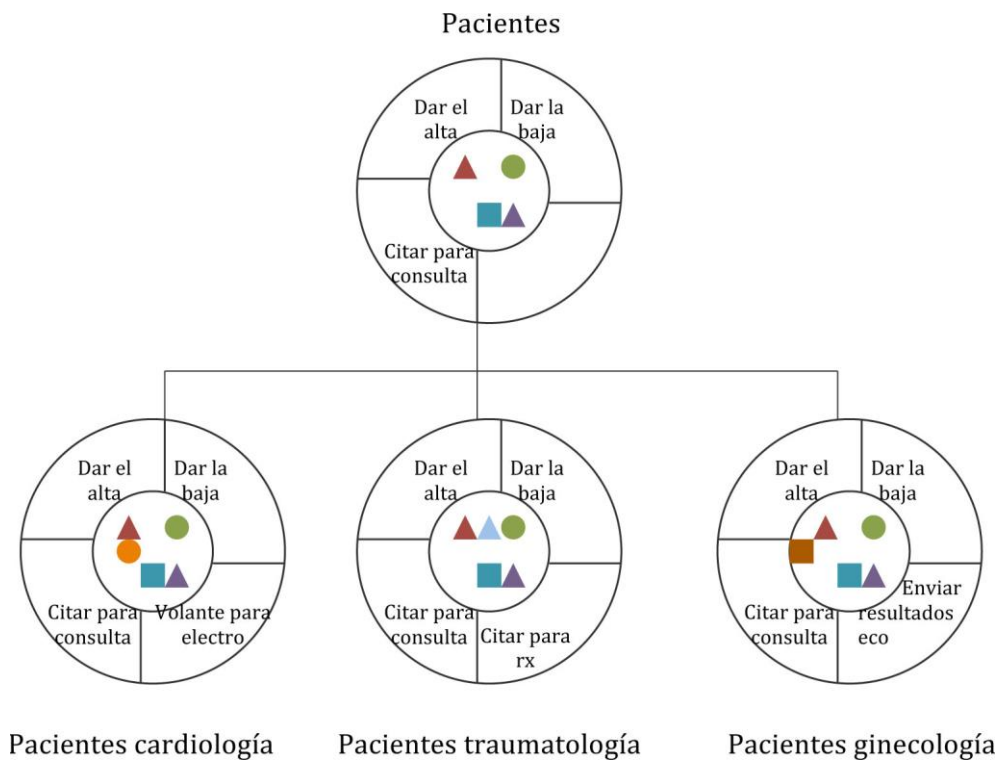


Figura 7.6: Representación de la herencia en programación orientada a objetos

Pueden hacerse tantos niveles de herencia como se desee. Por ejemplo, dentro de la clase `Animales`, se pueden definir las subclases `Reptiles`, `Mamíferos` y `Aves`. A su vez, dentro de la clase `Mamíferos`, puede definirse la clase `Humanos`, con las subclases `Hombre` y `Mujer` (Figura 7.7).

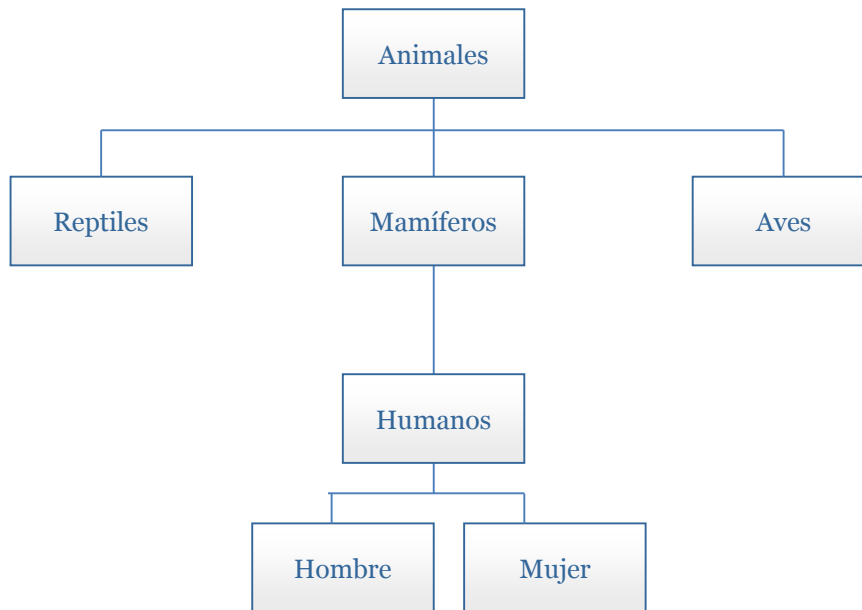


Figura 7.7: Representación de la herencia en varios niveles

Tipos de herencia:

En algunos lenguajes como Python o C++ hay dos tipos de herencia: simple y múltiple.

La **herencia simple** es aquella en la que la subclase solo hereda datos y métodos de una única superclase, mientras que en la **herencia múltiple** se puede heredar de más de una clase.

La herencia simple es la que se ha visto hasta ahora. La Figura 7.8 muestra un ejemplo de herencia múltiple de la clase Medios de transporte.

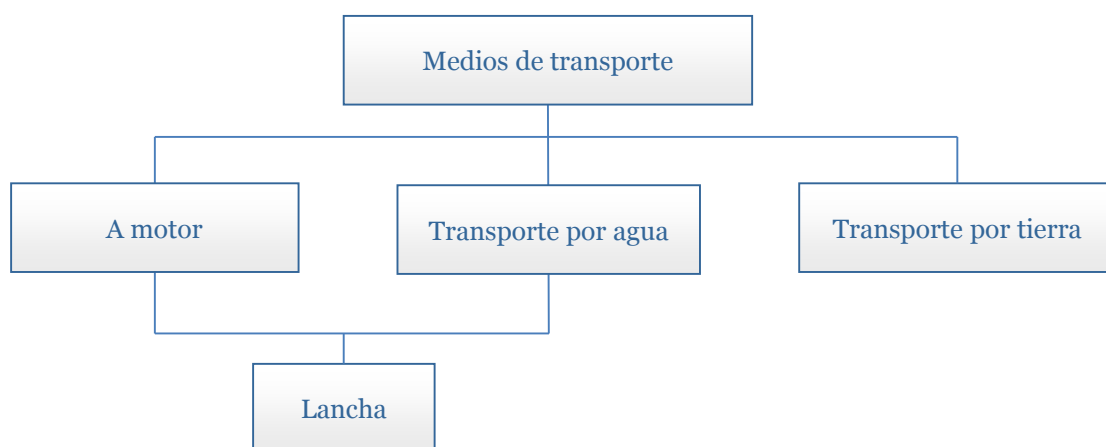


Figura 7.8: Representación de la herencia múltiple

Aunque la herencia múltiple puede ser muy útil para modelar ciertos problemas de la vida real, pueden surgir problemas, por ejemplo, cuando hay campos o métodos en las clases padre con el mismo nombre que en realidad son distintos. ¿Cuál se hereda? Aunque, lógicamente, es posible lidiar con esas ambigüedades, hay que ser muy cuidadoso en la escritura del código. Este tipo de herencias no van a trabajarse en este curso.

Polimorfismo

«El **polimorfismo** hace referencia a la habilidad que tienen los objetos de diferentes clases a responder a métodos con el mismo nombre, pero con implementaciones diferentes» (Fernández, 2012).

Esto también sucede en el mundo real, por ejemplo, una serpiente y un pájaro se pueden desplazar, aunque lo hacen de forma distinta. Esta ambigüedad se resuelve en Python de manera muy sencilla, para ver un ejemplo vamos a adelantar la sintaxis de las clases en Python. Sean las clases Pajaro y Serpiente:

```
class Pajaro():
    def desplazar(self):
        print("Volar")

class Serpiente():
    def desplazar(self):
        print("Reptar")
```

Si queremos llamar al método desplazar eliminando la ambigüedad, podemos definir la función:

```
def mover(animal):
    animal.desplazar()
```

Si ejecutamos

```
>>> p = Pajaro()
>>> s = Serpiente()
>>> p.desplazar()
Volar
>>> s.desplazar()
Reptar
>>> mover(p)
Volar
>>> mover(s)
Reptar
```

9.4. POO en Python

Una vez vistos los fundamentos de programación orientada a objetos, vamos a ver cómo se implementan en Python y qué particularidades tiene este lenguaje frente a otros.

Clases

Para definir una clase se usa la sentencia `class` seguida del nombre de la clase. Para realizar las tareas de inicialización hay dos métodos `__init__` y `__new__`. En este curso vamos a trabajar solo con `__init__`, que se utiliza para ejecutar todas las operaciones iniciales que sean necesarias. Por lo general, en los nombres de las clases, cada palabra comienza por mayúscula (por ejemplo, `Primera` o `PrimeraClase`). Esta notación se denomina *CamelCase*.

Así, podemos crear la clase

```
class PrimeraClase:
    def __init__(self):
        print("Primera Clase")
```

Podemos ejecutar la clase guardándola en un fichero (lo usual es que tenga el mismo nombre que la clase, pero todo en minúscula).

Si se ejecuta

```
>>> p = PrimeraClase()
Primera Clase
```

En la sintaxis de definición de la clase, se ha utilizado el parámetro `self` dentro del método `__init__`. Este parámetro tiene la referencia del objeto que ejecuta el método. En realidad `self` no es una palabra reservada, por lo que en realidad podría usarse cualquier otra, práctica muy poco aconsejable, ya que su uso está ampliamente difundido. Para invocar un método, no hace falta emplear la palabra `self`. Supongamos que a la clase `PrimeraClase` se le añade el método:

```
def nuevo(self):
    print("Nuevo metodo")
```

Y se ejecuta

```
>>> p = PrimeraClase()
Primera Clase
>>> p.nuevo()
Nuevo metodo
```

Atributos y objetos

Los atributos (también llamados en Python variables de instancia) se crean anteponiendo `self.` al nombre.

Supongamos que queremos crear la clase `Coche` que tiene unos atributos `marca`, `modelo`, `color`:

```
class Coche():
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color
```

Una vez creada la clase, se pueden crear los objetos

```
>>> v1 = Coche("Volkswagen", "Golf", "Azul")
>>> v2 = Coche("Toyota", "Yaris", "Blanco")
```

Se puede acceder al estado de las variables (recordemos que en Python no hay objetos y métodos privados) ejecutando, por ejemplo

```
>>> v1.marca
'Volkswagen'
```

Supongamos que en el programa principal ya existe una variable `marca`. Aunque en Python sea posible acceder a los objetos desde fuera de la clase, no se produce confusión entre nombres.

Para comprobarlo, se añade a la clase Coche el método:

```
def obtener_marca(self):
    marca = "marca en programa principal"
    print(self.marca)
    print(marca)
```

Y al ejecutar

```
>>> v1 = Coche("Volkswagen", "Golf", "Azul")
>>> v1.obtener_marca()
Volkswagen
marca en programa principal
```

Métodos de distancia

Los métodos de instancia (generalmente llamados métodos), son aquellos que se han definido en secciones anteriores: son las funciones que operan sobre los objetos de una clase. Aunque existen otros tipos de métodos, estos son los más comunes.

En el ejemplo anterior, `obtener_marca`, es un método de instancia. Veamos otro ejemplo, en el que el método recibe otros parámetros además de `self`. Añadimos a la clase `Coche` el método

```
def acelerar(self, km):
    print("acelerando {0} km".format(km))

>>> v1.acelerar(10)
acelerando 10 km
```

Puede verse que un método de instancia se define igual que una función, salvo por que en el método el primer parámetro debe hacer referencia siempre a la instancia que lo invoca (`self`).

Visibilidad

Una de las peculiaridades de Python con respecto a otros lenguajes orientados a objetos, como C++ o Java, se debe a la visibilidad de objetos y métodos de una clase. En Python no se pueden establecer métodos o atributos privados, son todos públicos. A pesar de que no se puede impedir el acceso desde fuera de la clase, existen maneras de indicar que no debería hacerse.

La primera, consiste en anteponer un guion bajo en el nombre de lo que se quiera hacer privado. Hay que tener cuidado, ya que esta notación no impide el acceso (como vamos a ver a continuación), simplemente funciona como recordatorio. Sea la clase

```
class PruebaVisibilidad():
    def _privado(self):
        print("Metodo privado")
```

Desde el intérprete se ejecuta

```
>>> t = PruebaVisibilidad()
>>> t._privado()
Metodo privado
```

Una forma más segura de proceder sería utilizar el doble espacio al principio del nombre. Por ejemplo, sea la clase

```
class Pseudoprivado():
    def __init__(self):
        self.__atributo = 3
```

Si se ejecuta

```
>>> p = Pseudoprivado()
>>> p.__atributo
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    p.__atributo
AttributeError: 'Pseudoprivado' object has no attribute '__atributo'
```


Este mecanismo, denominado *name mangling*, fue desarrollado, no para restringir la visibilidad, sino para evitar que una clase hija sobrescribiera accidentalmente los métodos de su clase padre. Para acceder al atributo `__atributo` hay que llamarlo `_Pseudoprivado__atributo`, por tanto

```
>>> p._Pseudoprivado__atributo
3
```

Este mecanismo no define atributos privados propiamente dichos, por eso pueden llamarse pseudoprivados.

Métodos especiales

Existen unos métodos especiales en Python que están contenidos en todas las clases que se crean. Son los llamados métodos especiales. En realidad toda clase que se crea es una subclase de una clase especial llamada `object`, que contiene los métodos especiales. Como ocurre con cualquier tipo de herencia, los métodos especiales pueden modificarse. Algunos de ellos son:

`__init__` Es el método especial más importante. Como ya hemos visto, inicializa un objeto cuando se crea. Necesita como parámetro la referencia a la instancia (`self`).

`__new__` Este parámetro no está presente en muchos lenguajes. Por lo general, basta con utilizar `__init__`. `__new__` se emplea cuando se quiere modificar un objeto inmutable (cadena, tupla) heredado en una subclase.

`__str__` Es el método al que invoca la función `str`. Devuelve una cadena con los caracteres que se quieren mostrar.

`__eq__` Se utiliza para determinar si dos variables son iguales. Necesita dos parámetros, `self` y la variable con que lo queremos comparar.

Herencia

La herencia es una cualidad esencial en el paradigma orientado a objetos, que permite reutilizar código.

Herencia simple:

Veamos la sintaxis de la herencia simple en Python con un ejemplo:

```
class Padre():
    def __init__(self):
        self.x = 8
        print("Clase padre")
    def metodo(self):
        print("Metodo de la clase padre")

class Hija(Padre):
    def met(self):
        print("Metodo de la clase hija")

>>> h = Hija()
Clase padre
>>> h.metodo()
Metodo de la clase padre
>>> h.met()
Metodo de la clase hija
>>> h.x
8
```

Como puede verse, para crear una clase hija, solo hay que declarar quién es el padre en la sentencia `class`. Tampoco es necesario invocar al método `__init__` ya que cuando se crea la hija, se llama al padre y se ejecuta allí. No obstante, si se quiere añadir, se inicializará en la clase hija, obteniendo el resultado

```
class Hija(Padre):
    def __init__(self):
        print("Clase hija")
    def met(self):
        print("Metodo de la clase hija")

>>> h = Hija()
Clase hija
```

Herencia múltiple

Python también admite las herencias múltiples, es decir, subclases con más de una superclase. La sintaxis es muy similar a la anterior

```
class Padre1():
    def metodo1(self):
        print("Metodo de la clase padre 1")
class Padre2():
    def metodo2(self):
        print("Metodo de la clase padre 2")
class Hija(Padre1, Padre2):
    def met(self):
        print("Metodo de la clase hija")

>>> h = Hija()
>>> h.metodo1()
Metodo de la clase padre 1
>>> h.metodo2()
Metodo de la clase padre 2
>>> h.met()
Metodo de la clase hija
```

Como ya se ha dicho en secciones previas, las herencias múltiples pueden dar lugar a ambigüedades, como el problema del diamante. Supongamos que se tiene una herencia múltiple como la representada en la Figura 7.9

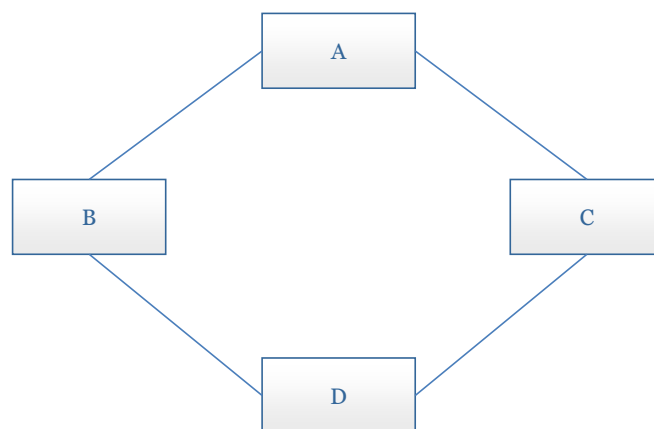


Figura 7.9: Representación del problema del diamante

El problema del diamante recibe su nombre por la forma de su representación. La ambigüedad se produce siempre que haya un método definido en A que se invoca desde D. ¿D lo hereda de B o de C?

Python resuelve el problema creando una lista de clases que se buscan de izquierda a derecha y de abajo a arriba. A continuación, se eliminan todas las clases repetidas menos la última. En este caso, Python crearía la lista D, B, A, C, A qué pasará a ser D, B, A, C.

9.5. Referencias bibliográficas

Fernández, A. (2012). *Python 3 al descubierto*. Madrid: Ed. RC Libros.

Joyane, L. (2008). *Fundamentos de la programación. Algoritmos y Estructura de Datos (4ª Ed)*. Madrid: McGraw-Hill.

Lo + recomendado

No dejes de leer...

Ejercicios de clases en Python

En este enlace se explican los conceptos básicos de POO en Python así como un ejercicio resuelto.

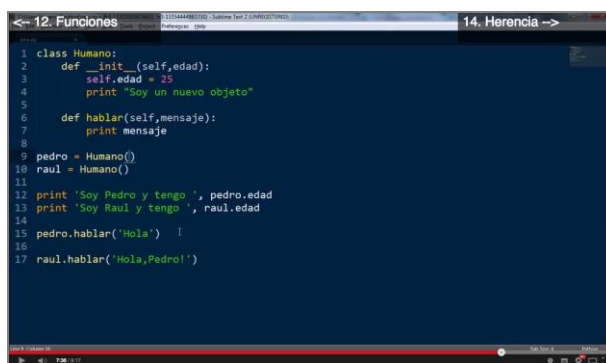
Accede al artículo a través del aula virtual o desde la siguiente dirección web:

<http://pythonya.appspot.com/detalleconcepto?deta=Declaraci%C3%B3n%20de%20una%20clase%20y%20creaci%C3%B3n%20de%20objetos>

No dejes de ver...

Definición de clases, métodos y atributos en Python

En este enlace se puede cómo se definen clases en Python.

A screenshot of a video player interface. The video frame shows a Python code editor with the following code:

```
1 class Humano:
2     def __init__(self, edad):
3         self.edad = edad
4         print "Soy un nuevo objeto"
5
6     def hablar(self, mensaje):
7         print mensaje
8
9 pedro = Humano()
10 raul = Humano()
11
12 print 'Soy Pedro y tengo ', pedro.edad
13 print 'Soy Raul y tengo ', raul.edad
14
15 pedro.hablar('Hola')
16
17 raul.hablar('Hola, Pedro!')
```

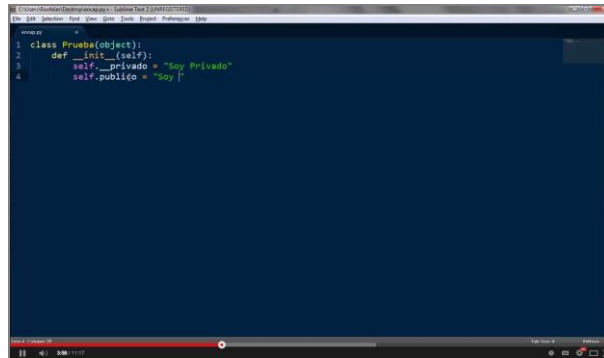
The video player has a progress bar at the bottom and navigation controls.

Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

<https://www.youtube.com/watch?v=VYXdjCZojA>

Encapsulación

En este vídeo se muestra qué hacer para conseguir limitar el acceso a métodos en Python.

A screenshot of a video player window. The video content shows a Python code editor with a dark blue background. The code defines a class named 'Prueba' that inherits from 'object'. It has an '__init__' method that initializes two attributes: 'self._privado' with the value 'Soy Privado' and 'self.publico' with the value 'Soy'. The video player interface includes a progress bar at the bottom and a menu bar at the top.

```
1 class Prueba(object):  
2     def __init__(self):  
3         self._privado = "Soy Privado"  
4         self.publico = "Soy"
```

Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

<https://www.youtube.com/watch?v=OrS7nrbwgiA>

+ Información

A fondo

Data model

Documentación de Python relativa a la POO.

Accede al documento a través del aula virtual o desde la siguiente dirección web:

<https://docs.python.org/3/reference/datamodel.html>

Advanced special class methods

Explica algunos métodos especiales de Python.

Accede al documento a través del aula virtual o desde la siguiente dirección web:

http://www.diveintopython.net/object_oriented_framework/special_class_methods2.html

Inmersión al modo interactivo de Python

Leonardo, J & Caballero, G. (31 de diciembre de 2013). *Inmersión al modo interactivo Python*. Recuperado de <https://plone-spanish-docs.readthedocs.org/es/latest/index.html>

En este artículo se explica cómo examinar los módulos en memoria.

Accede al documento a través del aula virtual o desde la siguiente dirección web:

https://plone-spanish-docs.readthedocs.org/es/latest/python/una_pequena_inmersion_python.html

Sobrecarga de operadores

Se explica en qué consiste la sobrecarga de operadores.

Accede al documento a través del aula virtual o desde la siguiente dirección web:

<http://pensandocomoprogramador.blogspot.com.es/2012/08/148-sobrecarga-de-operadores.html>

Test

1. La programación orientada a objetos:
 - A. Se centra en el algoritmo.
 - B. Es equivalente a la programación estructurada.
 - C. Se centra en el diseño de formato datos.

2. Pueden ser objetos:
 - A. Solo los objetos tangibles.
 - B. Animales, objetos espaciales, organizaciones, etc.
 - C. Sumar, imprimir, calcular.

3. El estado de un objeto:
 - A. Viene determinado por la clase a la que pertenece.
 - B. Son los datos que caracterizan al objeto.
 - C. Es el conjunto de métodos definidos en una clase.

4. Cuando un objeto A envía un mensaje a otro objeto B para que realice una acción determinada:
 - A. El mensaje tiene que incluir cómo quiere A que se realicen las tareas.
 - B. El mensaje no tiene que incluir cómo quiere A que se realicen las tareas.
 - C. Puede especificarse cómo se quieren las tareas, aunque no es obligatorio.

5. La descripción de un conjunto de objetos, es:
 - A. Un método.
 - B. Un mensaje.
 - C. Una clase.

6. Si una clase A es hija de una clase B :
 - A. A hereda las características de B y puede añadir atributos o métodos nuevos, pero no modificar los heredados.
 - B. A hereda las características de B y puede añadir atributos o métodos nuevos, así como modificar los heredados.
 - C. A hereda las características de B y puede añadir solo atributos nuevos, pero no métodos.

7. La habilidad que tienen los objetos de diferentes clases a responder a métodos con el mismo nombre, pero con implementaciones diferentes recibe el nombre de:

- A. Polimorfismo.
- B. Herencia múltiple.
- C. Herencia simple.

8. El parámetro `self` se incluye como parámetro en las clases porque:

- A. Es obligatorio al tratarse de una palabra reservada.
- B. Es recomendable ya que su uso está muy extendido.
- C. Es una cuestión poco relevante ya que en la práctica no se emplea casi nunca.

9. La sintaxis de un método:

- A. Es prácticamente igual a la sintaxis de una función.
- B. Es prácticamente igual a la sintaxis de una variable.
- C. Es prácticamente igual a la sintaxis de un atributo.

10. En Python:

- A. No se pueden modificar los datos de una clase con funciones que no pertenezcan a la clase.
- B. Solo se pueden modificar los datos públicos de una clase con funciones que no pertenezcan a la clase.
- C. Los datos de todas las clases son públicos.