Programación orientada a objetos

Ricardo Pérez López

IES Doñana, curso 2020/2021

Índice general

1. Introducción

1.1. Recapitulación

Recordemos lo que hemos aprendido hasta ahora:

- La **abstracción de datos** nos permite definir tipos de datos complejos llamados **tipos abstractos de datos** (*TAD*), que se representan únicamente mediante las **operaciones** que manipulan esos datos y con **independencia de su implementación**.
- Las funciones pueden tener **estado interno** usando funciones de orden superior y variables no locales.
- Una función puede representar un dato.
- Un dato puede tener estado interno usando el estado interno de la función que lo representa.

Además:

- El **paso de mensajes** agrupa las operaciones que actúan sobre ese dato dentro de una función que responde a diferentes mensajes **despachando** a otras funciones dependiendo del mensaje recibido.
- La función que representa al dato **encapsula su estado interno junto con las operaciones** que lo manipulan en *una sola unidad sintáctica* que oculta sus detalles de implementación.

En conclusión:

> Una función, por tanto, puede implementar un tipo abstracto de datos.

1.2. La metáfora del objeto

Al principio, distinguíamos entre funciones y datos: las funciones realizan operaciones sobre los datos y éstos esperan pasivamente a que se opere con ellos.

Cuando empezamos a representar a los datos con funciones, vimos que los datos también pueden encapsular **comportamiento**.

Esos datos ahora representan información, pero también **se comportan** como las cosas que representan.

Por tanto, los datos ahora saben cómo reaccionar ante los mensajes que reciben cuando el resto del programa les envía mensajes.

Esta forma de ver a los datos como objetos activos que interactúan entre sí y que son capaces de reaccionar y cambiar su estado interno en función de los mensajes que reciben, da lugar a todo un nuevo paradigma de programación llamado **orientación a objetos** o **programación orientada a objetos**.

> **Definición:**
>
> La **programación orientada a objetos** es un paradigma de programación en el que los programas son vistos como formados por entidades llamadas **objetos** que recuerdan su propio **estado in-**

terno y que se comunican entre sí mediante el **paso de mensajes** que se intercambian con la finalidad de:

- cambiar sus estados internos,
- compartir información y
- solicitar a otros objetos el procesamiento de dicha información.

La **programación orientada a objetos** (también llamada **OOP**, del inglés *Object-Oriented Programming*) es un método para organizar programas que reúne muchas de las ideas vistas hasta ahora.

Al igual que las funciones en la abstracción de datos, los objetos imponen barreras de abstracción entre el uso y la implementación de los datos.

Al igual que los diccionarios y funciones de despacho, los objetos responden a peticiones que otros objetos le hacen en forma de mensajes para que se comporte de determinada manera.

Los objetos tienen un estado interno local al que no se puede acceder directamente desde el entorno global, sino que debe hacerse por medio de las operaciones que proporciona el objeto.

A efectos prácticos, por tanto, los objetos son datos abstractos.

El sistema de objetos de Python proporciona una sintaxis cómoda para promover el uso de estas técnicas de organización de programas.

Gran parte de esta sintaxis se comparte entre otros lenguajes de programación orientados a objetos.

Ese sistema de objetos ofrece algo más que simple comodidad:

- Proporciona una **nueva metáfora** para diseñar programas en los que varios agentes independientes **interactúan** dentro del ordenador.
- Cada objeto **agrupa (*encapsula*)** el estado local y el comportamiento de una manera que abstrae la complejidad de ambos.
- Los objetos **se comunican entre sí** y se obtienen resultados útiles como consecuencia de su interacción.
- Los objetos no sólo transmiten mensajes, sino que también **comparten el comportamiento** entre otros objetos del mismo tipo y **heredan características** de otros tipos relacionados.

El paradigma de la programación orientada a objetos tiene su propio vocabulario que apoya la metáfora del objeto.

1.3. Perspectiva histórica

1.4. Lenguajes orientados a objetos

2. Clases y objetos

2.1. Clases

Una **clase** es una construcción sintáctica que los lenguajes de programación orientados a objetos proporcionan como *azúcar sintáctico* para **implementar tipos abstractos de datos** de una forma cómoda y directa sin necesidad de usar funciones de orden superior, estado local o diccionarios de despacho.

En programación orientada a objetos:

- Se habla siempre de **clases** y no de *tipos abstractos de datos*.
- Una **clase** es la **implementación de un tipo abstracto de datos**.
- Las clases definen **tipos de datos** de pleno derecho en el lenguaje de programación.

Recordemos el ejemplo del tema anterior en el que implementamos el tipo abstracto de datos **Depósito** mediante la siguiente **función**:

```
def deposito(fondos):
    def retirar(cantidad):
        nonlocal fondos
        if cantidad > fondos:
            return 'Fondos insuficientes'
        fondos -= cantidad
        return fondos

    def ingresar(cantidad):
        nonlocal fondos
        fondos += cantidad
        return fondos

    def saldo():
        return fondos

    def despacho(mensaje):
        if mensaje == 'retirar':
            return retirar
        elif mensaje == 'ingresar':
            return ingresar
        elif mensaje == 'saldo':
            return saldo
        else:
            raise ValueError('Mensaje incorrecto')

    return despacho
```

Ese mismo TAD se puede implementar como una **clase** de la siguiente forma:

```
class Deposito:
    def __init__(self, fondos):
        self.fondos = fondos

    def retirar(self, cantidad):
        if cantidad > self.fondos:
            return 'Fondos insuficientes'
        self.fondos -= cantidad
        return self.fondos

    def ingresar(self, cantidad):
        self.fondos += cantidad
        return self.fondos

    def saldo(self):
        return self.fondos
```

En el momento en que se ejecute esta definición, el intérprete incorporará al sistema un nuevo tipo

llamado `Deposito`.

Más tarde estudiaremos los detalles técnicos que diferencian ambas implementaciones, pero ya apreciamos que por cada operación sigue habiendo una función (aquí llamada **método**), que desaparece la función `despacho` y que aparece una extraña función `__init__`.

La definición de una clase es una estructura sintáctica que crea su propio ámbito y que está formada por una secuencia de sentencias que se ejecutarán cuando la ejecución del programa alcance esa definición:

```
class ⟨nombre⟩:
    ⟨sentencia⟩+
```

Todas las definiciones que se hagan dentro de la clase serán **locales** a ella, al encontrarse dentro del ámbito de dicha clase.

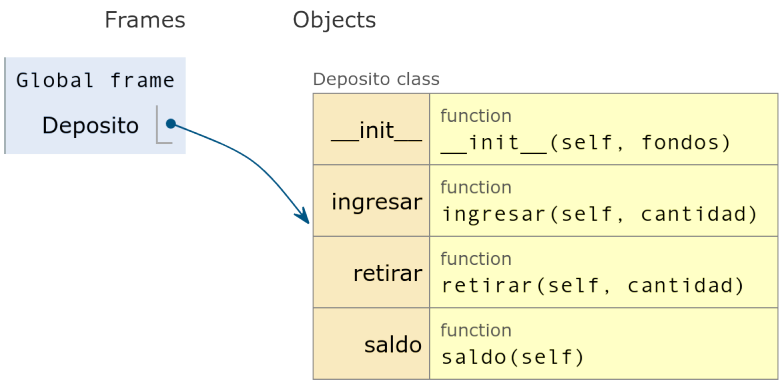
Por ello, las funciones definidas dentro de una clase pertenecen a dicha clase.

Por ejemplo, las funciones `__init__`, `retirar`, `ingresar` y `saldo` son locales a la clase `Deposito` y sólo existen dentro de ella.

Las funciones definidas dentro de una clase se denominan **métodos**.

Si ejecutamos la anterior definición en el Pythontutor, observaremos que se crea en memoria una estructura similar al **diccionario de despacho** que creábamos antes a mano, y que asocia el nombre de cara operación con la función (el método) correspondiente.

Esa estructura se liga al nombre de la clase en el marco del ámbito donde se haya declarado la clase (normalmente será el marco global).



La clase `Deposito` en memoria

2.2. Objetos

Un **objeto** representa un **dato abstracto** de la misma manera que una *clase* representa un *tipo abstracto de datos*.

Es decir: un objeto es un caso particular de una clase, motivo por el que también se le denomina ***instancia* de una clase**.

Un objeto es **un dato que pertenece al tipo definido por la clase** de la que es instancia.

También se puede decir que «**el objeto *pertenece* a la clase**» aunque sea más correcto decir que «**es *instancia* de la clase**».

El proceso de crear un objeto a partir de una clase se denomina **instanciar la clase** o **instanciación**.

En un lenguaje orientado a objetos *puro*, todos los datos que manipula el programa son objetos y, por tanto, instancias de alguna clase.

Existen lenguajes orientados a objetos *impuros* o *híbridos* en los que coexisten objetos con otros datos que no son instancias de clases.

Python es considerado un lenguaje orientado a objetos **puro**, ya que en Python todos los datos son objetos.

Por ejemplo, en Python:

- El tipo `int` es una clase.
- El entero `5` es un objeto, instancia de la clase `int`.

Java es un lenguaje orientado a objetos **impuro**, ya que un programa Java manipula objetos pero también manipula otros datos llamados *primitivos*, que no son instancias de ninguna clase sino que pertenecen a un *tipo primitivo* del lenguaje.

Por ejemplo, en Java:

- El tipo `String` es una clase, por lo que la cadena `"Hola"` es un objeto, instancia de la clase `String`.
- El tipo `int` es un tipo primitivo del lenguaje, por lo que el número `5` no es ningún objeto, sino un dato primitivo.

Las clases, por tanto, son como *plantillas* para crear objetos que comparten el mismo comportamiento y (normalmente) la misma estructura interna.

En Python podemos instanciar una clase (creando así un nuevo objeto) llamando a la clase como si fuera una función, del mismo modo que hacíamos con la implementación funcional que hemos estado usando hasta ahora:

```
>>> dep = Deposito(100)
>>> dep
<__main__.Deposito object at 0x7fba5a16d978>
```

Para saber la clase a la que pertenece el objeto, se usa la función `type` (recordemos que en Python todos los tipos son clases):

```
>>> type(dep)
<class '__main__.Deposito'>
```

Se nos muestra que la clase del objeto `dep` es `__main__.Deposito`, que representa la clase `Deposito` definida en el módulo `__main__`.

2.3. Estado

Los objetos son datos abstractos que poseen su propio estado interno, el cual puede cambiar durante la ejecución del programa como consecuencia de los mensajes recibidos o enviados por los objetos.

Eso significa que **los objetos son datos mutables**.

Dos objetos distintos podrán tener estados internos distintos.

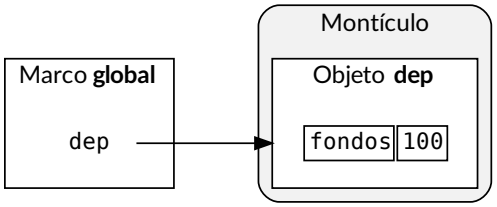
2.3.1. Atributos

Las variables de estado que almacenan el estado interno del objeto se denominan, en terminología orientada a objetos, **atributos**, **campos**, **propiedades** o **variables de instancia** del objeto.

Los atributos se implementan como *variables locales* al objeto.

Cuando se crea un objeto, se le asocia en el montículo una zona de memoria que almacena los atributos del objeto de forma similar al diccionario que usan las clases para almacenar sus definiciones locales.

Esa estructura en forma de diccionario representa al objeto dentro de la memoria («*es*» el objeto) y asocia el nombre de cada atributo con el valor que tiene dicho atributo en ese objeto.



Objeto `dep` y su atributo `fondos`

En Python es posible acceder directamente al estado interno de un objeto (o, lo que es lo mismo, al valor de sus atributos), cosa que, en principio, podría considerarse una violación del principio de ocultación de información y del concepto mismo de abstracción de datos.

Incluso es posible cambiar directamente el valor de un atributo desde fuera del objeto, o crear atributos nuevos dinámicamente.

Todo esto puede resultar chocante para un programador de otros lenguajes, pero en la práctica resulta útil al programador por la naturaleza dinámica del lenguaje Python y por el estilo de programación que promueve.

En Python, la única forma de acceder a un atributo de un objeto es usando la *notación punto*:

objeto . *atributo*

Por ejemplo, para acceder al atributo `fondos` de un objeto `dep` de la clase `Deposito`, se usaría la expresión `dep.fondos`:

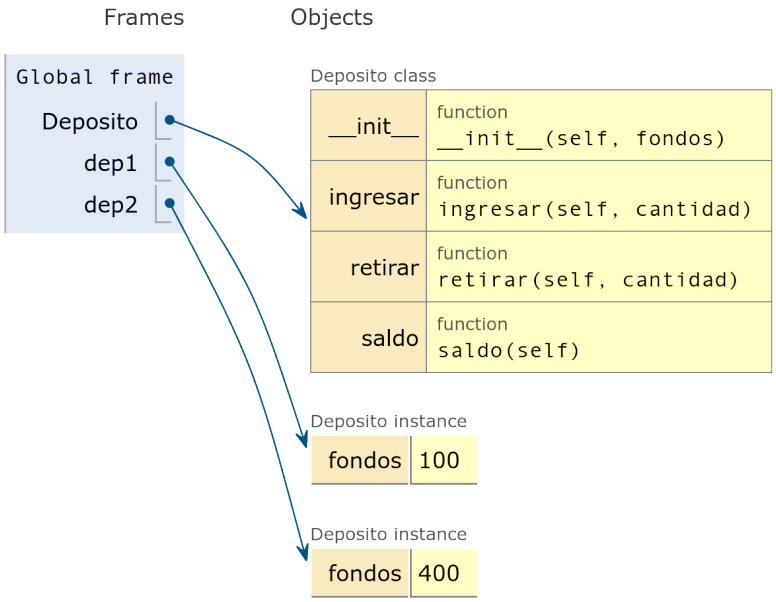
```
>>> dep = Deposito(100)
>>> dep.fondos
100
```

Y podemos cambiar el valor del atributo mediante asignación (cosa que, en general, no resultaría aconsejable):

```
>>> dep.fondos = 400
>>> dep.fondos
400
```

Por supuesto, dos objetos distintos pueden tener valores distintos en sus atributos:

```
>>> dep1 = Deposito(100)
>>> dep2 = Deposito(400)
>>> dep1.fondos          # el atributo fondos del objeto dep1 vale 100
100
>>> dep2.fondos          # el mismo atributo en el objeto dep2 vale 400
400
```



La clase `Deposito` y los objetos `dep1` y `dep2` en memoria

Como cualquier variable en Python, un atributo empieza a existir en el momento en el que se le asigna un valor:

```
>>> dep.otro
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Deposito' object has no attribute 'otro'
>>> dep.otro = 'hola'
>>> dep.otro
'hola
```

Por tanto, en Python los atributos de un objeto se crean en tiempo de ejecución mediante una simple asignación.

Este comportamiento contrasta con el de otros lenguajes de programación, como por ejemplo en Java, donde los atributos de un objeto vienen determinados de antemano por la clase a la que pertenece y siempre son los mismos.

Es decir: en Java, dos objetos de la misma clase siempre tendrán los mismos atributos, definidos por la clase (aunque el mismo atributo puede tener valores distintos en ambos objetos, naturalmente).

Ese comportamiento dinámico de Python a la hora de crear atributos permite resultados interesantes imposibles de conseguir en Java, como que dos objetos distintos de la misma clase puedan poseer distintos atributos:

```
>>> dep1 = Deposito(100)
>>> dep2 = Deposito(400)
>>> dep1.uno = 'hola'       # el atributo uno sólo existe en dep1
>>> dep2.otro = 'adiós'     # el atributo otro sólo existe en dep2
>>> dep1.uno
'hola'
>>> dep2.uno
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Deposito' object has no attribute 'uno'
>>> dep2.otro
'adiós'
>>> dep1.otro
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Deposito' object has no attribute 'otro'
```

Con Pythontutor podemos observar lo que ocurre al instanciar dos objetos y crear atributos distintos en cada objeto:

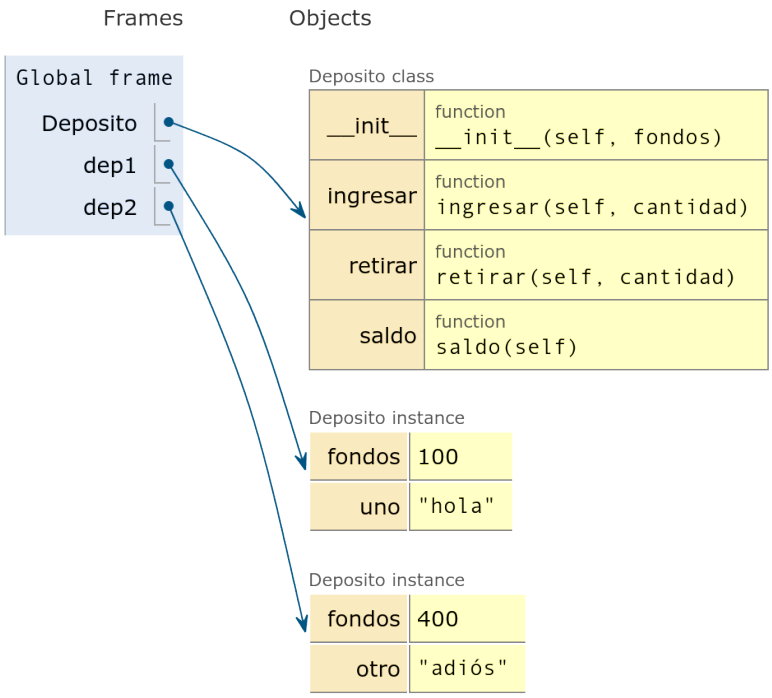
```
class Deposito:
    def __init__(self, fondos):
        self.fondos = fondos

    def retirar(self, cantidad):
        if cantidad > self.fondos:
            return 'Fondos insuficientes'
        self.fondos -= cantidad
        return self.fondos

    def ingresar(self, cantidad):
        self.fondos += cantidad
        return self.fondos
```

```
    def saldo(self):
        return self.fondos

dep1 = Deposito(100)
dep2 = Deposito(400)
dep1.uno = 'hola'
dep2.otro = 'adiós'
```



La clase `Deposito` y los objetos `dep1` y `dep2` con distintos atributos

2.4. Referencias

Cuando se ejecuta este código:

```
>>> dep = Deposito(100)
```

lo que ocurre es lo siguiente:

1. Se crea en el montículo una instancia de la clase `Deposito`, representada por una estructura con forma de diccionario.
2. Se invoca al método `__init__` sobre el objeto recién creado (ya hablaremos de ésto más adelante).

3. La expresión `Deposito(100)` devuelve una **referencia** al nuevo objeto, que representa, a grandes rasgos, la posición de memoria donde se encuentra almacenado el objeto.
4. Esa referencia es la que se almacena en la variable `dep`. Es decir: **en la variable no se almacena el objeto como tal, sino una *referencia* al objeto**.

En este ejemplo, `0x7fba5a16d978` es la dirección de memoria donde está almacenado el objeto al que hace referencia la variable `dep`:

```
>>> dep
<__main__.Deposito object at 0x7fba5a16d978>
```

Cuando una variable contiene una referencia a un objeto, decimos que la variable **se refiere** al objeto o que **apunta** al objeto.

Aunque actualmente las referencias representan direcciones de memoria, eso no quiere decir que vaya a ser siempre así. Ese es un detalle de implementación basada en una decisión de diseño del intérprete que puede cambiar en posteriores versiones del mismo.

Esa decisión, en la práctica, es una cuestión que no nos afecta (o no debería, al menos) a la hora de escribir nuestros programas.

Con Pythontutor podemos observar las estructuras que se forman al declarar la clase y al instanciar dicha clase en un nuevo objeto:

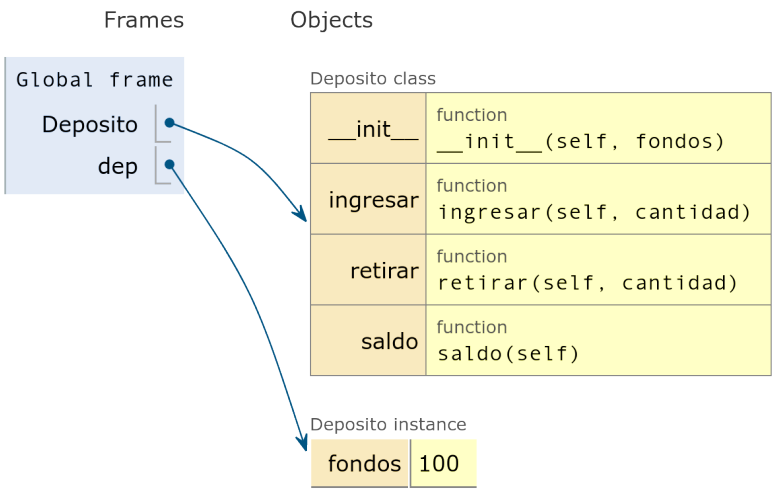
```
class Deposito:
    def __init__(self, fondos):
        self.fondos = fondos

    def retirar(self, cantidad):
        if cantidad > self.fondos:
            return 'Fondos insuficientes'
        self.fondos -= cantidad
        return self.fondos

    def ingresar(self, cantidad):
        self.fondos += cantidad
        return self.fondos

    def saldo(self):
        return self.fondos

dep = Deposito(100)
```



La clase `Deposito` y el objeto `dep` en memoria

Los objetos tienen existencia propia e independiente y permanecerán en la memoria siempre que haya al menos una referencia que apunte a él.

De hecho, un objeto puede tener varias referencias apuntándole.

Por ejemplo, si hacemos:

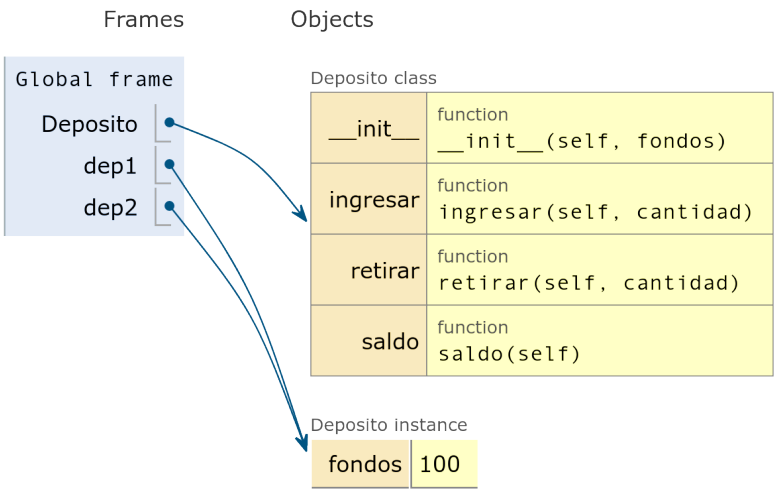
```
dep1 = Deposito(100)
dep2 = dep1
```

tendremos dos variables que contienen la misma referencia y, por tanto, ***se refieren* (o *apuntan*) al mismo objeto**.

Es exactamente el concepto de **alias de variables** que estudiamos en programación imperativa.

No olvidemos que las variables no contienen al objeto en sí mismo, sino una referencia a éste.

Gráficamente, el caso anterior se puede representar de la siguiente forma:



Dos variables (`dep1` y `dep2`) que *apuntan* al mismo objeto

2.4.1. Recolección de basura

En el momento en que un objeto se vuelva inaccesible (cosa que ocurrirá cuando no haya ninguna variable en el entorno que contenga una referencia a dicho objeto), el intérprete lo marcará como *candidato para ser eliminado*.

Cada cierto tiempo, el intérprete activará el **recolector de basura**, que es un componente que se encarga de liberar de la memoria a los objetos que están marcados como candidatos para ser eliminados.

Por tanto, el programador Python no tiene que preocuparse de gestionar manualmente la memoria ocupada por los objetos que componen su programa.

Por ejemplo:

```
dep1 = Deposito(100)  # crea el objeto y guarda una referencia a él en dep1
dep2 = dep1  # almacena en dep2 la referencia que hay en dep1
             # (a partir de ahora, ambas variables apuntan al mismo objeto)
del dep1     # elimina una referencia pero el objeto aún tiene otra
del dep2     # elimina la otra referencia y ahora el objeto es inaccesible
             # (cuando el recolector de basura se active, eliminará el objeto)
```

2.5. La antisimetría dato-objeto

Se da una curiosa contra-analogía entre los conceptos de dato y objeto:

- Los objetos ocultan sus datos detrás de abstracciones y exponen las funciones que operan con esos datos.
- Las estructuras de datos exponen sus datos y no contienen funciones significativas.

Son definiciones virtualmente opuestas y complementarias.

3. Paso de mensajes

3.1. Introducción

Como las clases implementan las operaciones como métodos, el paso de mensajes se realiza ahora invocando sobre el objeto el método correspondiente al mensaje que se enviaría al objeto.

Por ejemplo, si tenemos el objeto `dep` (una instancia de la clase `Deposito`) y queremos enviarle el mensaje `saldo` para saber cuál es el saldo actual de ese depósito, invocaríamos el método `saldo` sobre el objeto `dep` de esta forma:

```
>>> dep.saldo()
100
```

Si la operación requiere de argumentos, se le pasarán al método también:

```
>>> dep.retirar(25)
75
```

3.2. Ejecución de métodos

En Python, la ejecución de un método m con argumentos $a_1, a_2, \ldots, a_n$ sobre un objeto o que es instancia de la clase C tiene esta forma:

$$o.m(a_1, a_2, \ldots, a_n)$$

Y el intérprete lo traduce por una llamada a función con esta forma:

$$C.m(o, a_1, a_2, \ldots, a_n)$$

Es decir: el intérprete llama a la función m definida en la clase C y le pasa el objeto o como primer argumento (el resto de los argumentos originales irían a continuación).

Por ejemplo, hacer:

```
>>> dep.retirar(25)
```

equivale a hacer:

```
>>> Despacho.retirar(dep, 25)
```

De hecho, el intérprete traduce el primer código al segundo automáticamente.

Esto facilita la implementación del intérprete, ya que todo se convierte en llamadas a funciones.

3.3. Definición de métodos

Esa es la razón por la que los métodos se definen siempre con un parámetro extra que representa el objeto sobre el que se invoca el método (o, dicho de otra forma, el objeto que recibe el mensaje).

Ese parámetro extra (por regla de estilo) se llama siempre `self`, si bien ese nombre no es ninguna palabra clave y se podría usar cualquier otro.

Por tanto, siempre que definamos un método, lo haremos como una función que tendrá siempre un parámetro extra que será siempre el primero de sus parámetros y que se llamará `self`.

Por ejemplo, en la clase `Deposito`, obsérvese que todos los métodos tienen `self` como primer parámetro:

```
class Deposito:
    def __init__(self, fondos):
        self.fondos = fondos

    def retirar(self, cantidad):
        if cantidad > self.fondos:
            return 'Fondos insuficientes'
        self.fondos -= cantidad
        return self.fondos

    def ingresar(self, cantidad):
        self.fondos += cantidad
        return self.fondos

    def saldo(self):
        return self.fondos
```

El método `saldo` de la clase `Deposito` recibe un argumento `self` que, durante la llamada al método, contendrá el objeto sobre el que se ha invocado dicho método:

```
def saldo(self):
    return self.fondos
```

En este caso, contendrá el objeto del que se desea conocer los fondos que posee.

Por tanto, dentro de `saldo`, accedemos a los fondos del objeto usando la expresión `self.fondos`, y ese es el valor que retorna el método.

Dentro del programa, la expresión `dep.saldo()` se traducirá como `Deposito.saldo(dep)`.

Es importante recordar que **el parámetro `self` se pasa automáticamente** durante la llamada al método y, por tanto, **no debemos pasarlo nosotros** o se producirá un error por intentar pasar más parámetros de los requeridos por el método.

El método `ingresar` tiene otro argumento además del `self`, que es la cantidad a ingresar:

```
def ingresar(self, cantidad):
    self.fondos += cantidad
    return self.fondos
```

En este caso, `self` contendrá el objeto en el que se desea ingresar la cantidad deseada.

Dentro del método `ingresar`, la expresión `self.fondos` representa el valor del atributo `fondos` del objeto `self`.

Por tanto, lo que hace el método es incrementar el valor de dicho atributo en el objeto `self`, sumándole la cantidad indicada en el parámetro.

Por ejemplo, la expresión `dep.ingresar(35)` se traducirá como `Deposito.ingresar(dep, 35)`. Por tanto, en la llamada al método, `self` valdrá `dep` y `cantidad` valdrá `35`.

3.4. Métodos *mágicos* y constructores

En Python, los métodos cuyo nombre empieza y termina por `__` se denominan **métodos mágicos** y tienen un comportamiento especial.

En concreto, el método `__init__` se invoca automáticamente cada vez que se instancia un nuevo objeto a partir de una clase.

Coloquialmente, se le suele llamar el **constructor** de la clase, y es el responsable de *inicializar* el objeto de forma que tenga un estado inicial adecuado desde el momento de su creación.

Entre otras cosas, el constructor se encarga de asignarle los valores iniciales adecuados a los atributos del objeto.

Ese método recibe como argumentos (además del `self`) los argumentos indicados en la llamada a la clase que se usó para instanciar el objeto.

Por ejemplo: en la clase `Deposito`, tenemos:

```
class Deposito:
    def __init__(self, fondos):
        self.fondos = fondos
    # ...
```

Ese método `__init__` se encarga de crear el atributo `fondos` del objeto que se acaba de crear (y que recibe a través del parámetro `self`), asignándole el valor del parámetro `fondos`.

Cuidado: no confudir la expresión `self.fondos` con `fondos`. La primera se refiere al atributo `fondos` del objeto `self`, mientras que la segunda se refiere al parámetro `fondos`.

Cuando se crea un nuevo objeto de la clase `Deposito`, llamando a la clase como si fuera una función, se debe indicar entre paréntesis (como argumento) el valor del parámetro que luego va a recibir el método `__init__` (en este caso, los fondos iniciales):

```
dep = Deposito(100)
```

La ejecución de este código produce el siguiente efecto:

1. Se crea en memoria una instancia de la clase `Deposito`.
2. Se invoca el método `__init__` sobre el objeto recién creado, de forma que el parámetro `self` recibe una referencia a dicho objeto y el parámetro `fondos` toma el valor `100`, que es el valor del argumento en la llamada a `Deposito(100)`.

   En la práctica, esto equivale a decir que la expresión `Deposito(100)` se traduce a *r*`.__init__(100)`, donde *r* es una referencia al objeto recién creado.
3. La expresión `Deposito(100)` devuelve la referencia al objeto.
4. Esa referencia es la que se almacena en la variable `dep`.

Comprobar el funcionamiento del constructor en Pythontutor.

En resumen: la expresión $C($`(`a_1`,`a_2`,...,`a_n`)` usada para crear una instancia de la clase *C* lleva a cabo las siguientes acciones:

1. Crea en memoria una instancia de la clase *C* y guarda en una variable temporal (llamémosla *r*, por ejemplo) una referencia al objeto recién creado.
2. Invoca a *r*`.__init__(`a_1`,`a_2`,...,`a_n`)`
3. Devuelve *r*.

En consecuencia, los argumentos que se indican al instanciar una clase se enviarán al método `__init__` de la clase, lo que significa que tendremos que indicar tantos argumentos (y del tipo apropiado) como espere el método `__init__`.

En caso contrario, tendremos un error:

```
>>> dep = Deposito()  # no indicamos ningún argumento cuando se espera uno
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 1 required positional argument: 'fondos'
>>> dep = Deposito(1, 2)  # mandamos dos argumentos cuando se espera sólo uno
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes 2 positional arguments but 3 were given
```

Es importante tener en cuenta, además, que **el constructor `__init__` no debe devolver ningún valor** (o, lo que es lo mismo, debe devolver `None`), o de lo contrario provocará un error de ejecución.

4. Identidad e igualdad

4.1. Identidad

Ya hemos dicho que los objetos tienen existencia propia e independiente.

La **identidad** describe la propiedad que tienen los objetos de distinguirse de los demás objetos.

> Dos objetos del mismo tipo son **idénticos** si un cambio en cualquiera de los objetos provoca también el mismo cambio en el otro objeto.

O dicho de otra forma: dos objetos son idénticos si son intercambiables en el código fuente del programa sin que se vea afectado el comportamiento de éste.

Es evidente que dos objetos de distinto tipo no pueden ser idénticos.

En el momento en que introducimos la mutabilidad en nuestro modelo computacional, muchos conceptos que antes eran sencillos se vuelven problemáticos.

Por ejemplo, consideremos el problema de determinar si dos cosas son «la misma cosa».

Supongamos que hacemos:

```
def restador(cantidad):
    def aux(otro):
        return otro - cantidad
    return aux
res1 = restador(25)
res2 = restador(25)
```

¿Son `res1` y `res2` la misma cosa? Una respuesta aceptable podría ser que sí, ya que tanto `res1` como `res2` se comportan de la misma forma (los dos son funciones que restan 25 a su argumento). De hecho, `res1` puede sustituirse por `res2` en cualquier lugar de un programa sin que cambie el resultado.

En cambio, supongamos que hacemos dos llamadas a `Deposito(100)`:

```
dep1 = Deposito(100)
dep2 = Deposito(100)
```

¿Son `dep1` y `dep2` la misma cosa? Evidentemente no, ya que al enviarles mensajes a uno y otro podemos obtener resultados distintos ante los mismos mensajes:

```
>>> dep1.retirar(20)
80
>>> dep1.saldo()
80
>>> dep2.saldo()
100
```

Incluso aunque podamos pensar que `dep1` y `dep2` son «iguales» en el sentido de que ambos han sido creados evaluando la misma expresión (`Deposito(100)`), no es verdad que podamos sustituir `dep1` por `dep2` en cualquier expresión sin cambiar el resultado de evaluar dicha expresión.

Es otra forma de decir que con los objetos no hay transparencia referencial, ya que se pierde en el momento en que incorporamos estado y mutabilidad en nuestro modelo computacional.

Pero al perder la transparencia referencial, la noción de lo que significa que dos objetos sean «el mismo objeto» se vuelve difícil de capturar de una manera formal. De hecho, el significado de «el mismo» en el mundo real que estamos modelando con nuetro programa es ya difícil de entender.

En general, sólo podemos determinar si dos objetos aparentemente idénticos son realmente «el mismo objeto» modificando uno de ellos y observando a continuación si el otro se ha cambiado de la misma forma.

Pero, ¿cómo podemos decir si un objeto ha «cambiado» si no es observando el «mismo» objeto dos veces y comprobando si ha cambiado alguna propiedad del objeto de la primera observación a la siguiente?

Por tanto, no podemos decir que ha habido un «cambio» sin alguna noción previa de «igualdad», y

no podemos determinar la igualdad sin observar los efectos del cambio.

Un ejemplo de cómo puede afectar este problema en programación, sería considerar el caso en que Pedro y Pablo tienen un depósito con 100 € cada uno. Hay una enorme diferencia entre definirlo así:

```
dep_Pedro = Deposito(100)
dep_Pablo = Deposito(100)
```

y definirlo así:

```
dep_Pedro = Deposito(100)
dep_Pablo = dep_Pedro
```

En el primer caso, los dos depósitos son distintos. Las operaciones realizadas por Pedro no afectarán a la cuenta de Pablo, y viceversa.

En el segundo caso, en cambio, hemos definido a `dep_Pablo` para que sea exactamente la misma cosa que `dep_Pedro`.

Por tanto, ahora Pedro y Pablo son cotitulares de un depósito compartido, y si Pedro hace una retirada de efectivo a través de `dep_Pedro`, Pablo observará que hay menos dinero en `dep_Pablo`.

Estas dos situaciones, similares pero distintas, pueden provocar confusión al crear modelos computacionales. Concretamente, con el depósito compartido puede ser especialmente confuso el hecho de que haya un objeto (el depósito) con dos nombres distintos (`dep_Pedro` y `dep_Pablo`). Si estamos buscando todos los sitios de nuestro programa donde pueda cambiarse el depósito de `dep_Pedro`, tendremos que recordar buscar también los sitios donde se cambie a `dep_Pablo`.

Con respecto a los anteriores comentarios sobre «igualdad» y «cambio», obsérvese que si Pedro y Pablo sólo pudieran comprobar sus saldos y no pudieran realizar operaciones que cambiaran los fondos del depósito, entonces no existiría el problema de comprobar si los dos depósitos son distintos.

En general, siempre que no se puedan modificar los objetos, podemos suponer que un objeto compuesto se corresponde con la totalidad de sus partes.

Por ejemplo, un número racional está determinado por su numerador y su denominador. Pero este punto de vista deja de ser válido cuando incorporamos mutabilidad, donde un objeto compuesto tiene una «identidad» que es algo distinto de las partes que lo componen.

Un depósito sigue siendo «el mismo» depósito aunque cambiemos sus fondos haciendo una retirada de efectivo. Igualmente, podemos tener dos depósitos distintos con el mismo estado interno.

Esta complicación es consecuencia, no de nuestro lenguaje de programación, sino de nuestra percepción del depósito bancario como un objeto. Por ejemplo, no tendría sentido para nosotros considerar que un número racional es un objeto mutable con identidad puesto que al cambiar su numerador ya no tenemos «el mismo» número racional.

Como usamos **referencias** para referirnos a un determinado objeto y acceder al mismo, resulta fácil comprobar si dos objetos son *idénticos* (es decir, si son el mismo objeto) comparando referencias. Si las referencias son iguales, es que estamos ante un único objeto.

Esto es así ya que, por lo general, las referencias se corresponden con direcciones de memoria. Es decir: una referencia a un objeto normalmente representa la dirección de memoria donde se empieza

a almacenar dicho objeto.

Dos objetos pueden ser **iguales** y, en cambio, no ser *idénticos*.

La forma de comprobar en Python si dos objetos son *idénticos* es usar el operador `is` que ya conocemos:

> La expresión ***o* `is` *p*** devolverá `True` si tanto *o* como *p* son referencias al mismo objeto.

Por ejemplo:

```
>>> dep1 = Deposito(100)
>>> dep2 = dep1
>>> dep1 is dep2
True
```

En cambio:

```
>>> dep1 = Deposito(100)
>>> dep2 = Deposito(100)
>>> dep1 is dep2
False
```

4.2. Igualdad

En el código anterior:

```
dep1 = Deposito(100)
dep2 = Deposito(100)
```

es evidente que `dep1` y `dep2` hacen referencia a objetos separados y, por tanto, **no son *idénticos***, ya que no se refieren *al mismo* objeto.

En cambio, sí podemos decir que **son *iguales*** ya que pertenecen a la misma clase, poseen el mismo estado interno y se comportan de la misma forma ante la recepción de la mismos mensajes en el mismo orden:

```
>>> dep1 = Deposito(100)
>>> dep2 = Deposito(100)
>>> dep1.ingresar(30)
130
>>> dep1.retirar(45)
85
>>> dep2.ingresar(30)
130
>>> dep2.retirar(45)
85
```

Sin embargo, si preguntamos al intérprete si son iguales, nos dice que no:

```
>>> dep1 == dep2
False
```

Esto se debe a que, en ausencia de otra definición de *igualdad* y **mientras no se diga lo contrario, dos objetos de clases *definidas por el programador* son iguales si son idénticos**.

Es decir: por defecto, *x* `==` *y* equivale a *x* `is` *y*.

Para cambiar ese comportamiento predeterminado, tendremos que indicar qué significa que dos instancias de nuestra clase son iguales.

Por ejemplo: ¿cuándo podemos decir que dos objetos de la clase `Deposito` son iguales?

En este caso es fácil: dos instancias de `Deposito` son iguales cuando tienen el mismo estado interno. O lo que es lo mismo: dos depósitos son iguales cuando tienen los mismos fondos.

Para implementar nuestra propia lógica de igualdad en nuestra clase, debemos definir en ella el método mágico `__eq__`.

Este método se invocará automáticamente cuando se hace una comparación con el operador `==` y el primer operando es una instancia de nuestra clase. El segundo operando se enviará como argumento en la llamda al método.

Dicho de otra forma:

- `dep1 == dep2` equivale a `dep1.__eq__(dep2)`, siempre que la clase de `dep1` tenga definido el método `__eq__`.
- En caso contrario, seguirá valiendo lo mismo que `dep1 is dep2`, como acabamos de ver.

No es necesario preocuparse por el operador `!=`, ya que Python 3 lo define automáticamente a partir del `==`.

Una posible implementación del método `__eq__` podría ser:

```
def __eq__(self, otro):
    if type(self) != type(otro):
        return NotImplemented   # no tiene sentido comparar objetos de distinto tipo
    return self.fondos == otro.fondos  # son iguales si tienen los mismos fondos
```

Se devuelve el valor especial `NotImplemented` cuando no tiene sentido comparar un objeto de la clase `Deposito` con un objeto de otro tipo.

Si introducimos este método dentro de la definición de la clase `Deposito`, tendremos el resultado deseado:

```
>>> dep1 = Deposito(100)
>>> dep2 = Deposito(100)
>>> dep1 == dep2
True
>>> dep1.retirar(30)
70
>>> dep1 == dep2
False
>>> dep2.retirar(30)
70
>>> dep1 == dep2
True
```

5. Encapsulación

5.1. Encapsulación

En programación orientada a objetos, decimos que los objetos están **encapsulados**.

La encapsulación es un concepto fundamental en programación orientada a objetos, aunque no pertenece exclusivamente a este paradigma.

Aunque es uno de los conceptos más importantes de la programación orientación a objetos, no hay un consenso general y universalmente aceptado sobre su definición.

Además, es un concepto relacionado con la abstracción y la ocultación de información, y a veces se confunde con estos, lo que complica aún más la cosa.

Nosotros vamos a estudiar la encapsulación como dos **mecanismos** distintos pero relacionados:

- Por una parte, la encapsulación es un **mecanismo** de los lenguajes de programación que permite que **los datos y las operaciones** que se puedan realizar sobre esos datos **se agrupen juntos en una sola *unidad sintáctica***.
- Por otra parte, la encapsulación es un **mecanismo** de los lenguajes de programación por el cual **sólo se puede acceder al interior de un objeto mediante las operaciones** que forman su **barrera de abstracción**, impidiendo acceder directamente a los datos internos del mismo y garantizando así la **protección de datos**.

  En definitiva, nos referimos a un mecanismo que garantiza que los objetos actúan como **datos abstractos**.

Vamos a ver cada uno de ellos con más detalle.

5.1.1. La encapsulación como mecanismo de agrupamiento

El mecanismo de las **clases** nos permite crear estructuras que **agrupan datos y operaciones en una misma unidad**.

Al instanciar esas clases, aparecen los **objetos**, que conservan esa misma característica de agrupar datos (estado interno) y operaciones en una sola cosa.

De esta forma, las operaciones acompañan a los datos allá donde vaya el objeto.

Por tanto, al pasar un objeto a alguna otra parte del programa, estamos también pasando las operaciones que se pueden realizar sobre ese objeto, o lo que es lo mismo, los mensajes a los que puede responder.

En un lenguaje de programación, llamamos **ciudadano de primera clase** (*first-class citizen*) a todo aquello que:

- Puede ser **pasado como argumento** de una operación.
- Puede ser **devuelto como resultado** de una operación.
- Puede ser **asignado** a una variable.

Los objetos se pueden manipular (por ejemplo, enviarles mensajes) a través de las *referencias*, y éstas se pueden pasar como argumento, devolver como resultado y asignarse a una variable.

Por tanto, **los objetos son *ciudadanos de primera clase***.

Por ejemplo, si definimos una función que calcula la diferencia entre los saldos de dos depósitos, podríamos hacer:

```
def diferencia(dep1, dep2):
    return dep1.saldo() - dep2.saldo()
```

Es decir:

- La función `diferencia` recibe como argumentos los dos depósitos (que son objetos), por lo que éstos son ciudadanos de primera clase.
- Los objetos encapsulan:
  - sus *datos* (su estado interno) y
  - sus *operaciones* (los mensajes a los que puede responder)

  juntos en una sola unidad sintáctica, a la que podemos acceder usando una sencilla referencia, como `dep1` o `dep2`.
- Para obtener el saldo no se usa una función externa al objeto, sino que se le pregunta a este a través de la operación `saldo` contenida dentro del objeto.

En resumen, decir que los objetos están encapsulados es decir que:

- Agrupan datos y operaciones en una sola unidad.
- Son ciudadanos de primera clase.
- Es posible manipularlos por completo usando simplemente una referencia.
- La referencia representa al objeto a todos los niveles.

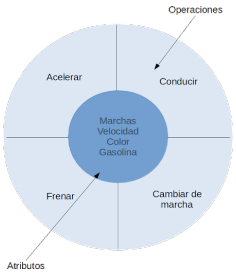
5.1.2. La encapsulación como mecanismo de protección de datos

Un dato abstracto es aquel que se define en función de las operaciones que se pueden realizar sobre él.

Los objetos son datos abstractos y, por tanto, su estado interno debería manejarse únicamente mediante operaciones definidas a tal efecto, impidiendo el acceso directo a los atributos internos del objeto.

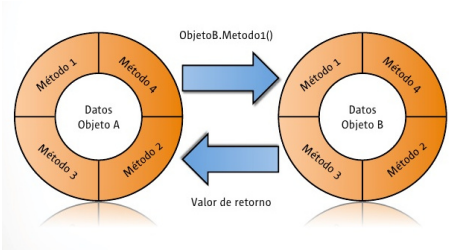
Según esto, podemos imaginar que:

- Los atributos que almacenan el estado interno del objeto están *encapsulados* dentro del mismo.
- Las operaciones con las que se puede manipular el objeto *rodean* a los atributos formando una *cápsula*, de forma que, para poder acceder al interior, hay que hacerlo necesariamente a través de esas operaciones.



Las operaciones forman una *cápsula*

Esas operaciones forman, efectivamente, la **interfaz** del dato abstracto y, por tanto, definen de qué manera podemos manipular al objeto desde el exterior del mismo.



Las operaciones forman una *cápsula*

5.1.2.1. Visibilidad

Para garantizar esta restricción de acceso, los lenguajes de programación a menudo facilitan un mecanismo por el cual el programador puede definir la **visibilidad** de cada miembro (atributo o método) de una clase.

De esta forma, el programador puede «marcar» que determinados atributos o métodos sólo sean accesibles desde el interior de esa clase o que, por el contrario, sí se pueda acceder a ellos desde el exterior de la misma.

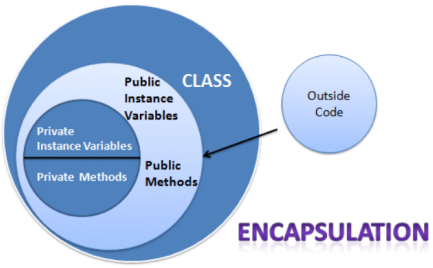
$$\text{Visibilidad} \begin{cases} \text{No se puede acceder desde el exterior, o} \\ \text{Sí se puede acceder desde el exterior} \end{cases}$$

Cada una de estas dos posibilidades da lugar a un tipo distinto de visibilidad:

- **Visibilidad *privada*:** si un miembro de una clase tiene visibilidad privada, sólo podrá accederse a él desde dentro de esa clase, pero no desde fuera de ella.
- **Visibilidad *pública*:** si un miembro de una clase tiene visibilidad pública, podrá accederse a él tanto desde dentro como desde fuera de la clase.

- Por tanto, **desde el exterior de un objeto sólo podremos acceder a los miembros marcados como *públicos* en la clase de ese objeto**.

El conjunto de los miembros públicos de una clase forman la ***interfaz* de la clase**, de forma similar a lo que ocurre con las interfaces de los tipos abstractos de datos.



Miembros *públicos* y *privados*

Cada lenguaje de programación tiene su propia manera de implementar el mecanismo de la visibilidad.

En Python, el mecanismo es muy sencillo:

Si el nombre de un miembro de una clase (atributo o método) empieza (pero no acaba) por `__`, entonces es *privado*. En caso contrario, es *público*.

Los *métodos mágicos* (como `__init__`, `__eq__`, etc.) tienen nombres que empiezan **y acaban** por `__`, así que no cumplen la condición anterior y, por tanto, son *públicos*.

Por ejemplo:

```
class Prueba:
    def __uno(self):
        print("Este método es privado, ya que su nombre empieza por __")

    def dos(self):
        print("Este método es público")

    def __tres__(self):
        print("Este método también es público, porque su nombre empieza y acaba por __")

p = Prueba()
p.__uno()    # No funciona, ya que __uno() es un método privado
p.dos()      # Funciona, ya que el método dos() es público
p.__tres__() # También funciona
```

Los miembros privados sólo son accesibles desde dentro de la clase:

```
>>> class Prueba:
...     def __uno(self):
...         print("Este método es privado, ya que su nombre empieza por __")
...
...     def dos(self):
...         print("Este método es público")
...         self.__uno() # Llama al método privado desde dentro de la clase
...
>>> p = Prueba()
>>> p.__uno()    # No funciona, ya que __uno() es un método privado
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Prueba' object has no attribute '__uno'
>>> p.dos()      # Funciona, ya que el método dos() es público
Este método es público
Este método es privado, ya que su nombre empieza por __
```

Con las variables de instancia ocurre exactamente igual:

```
>>> class Prueba:
...     def __init__(self, x):
...         self.__x = x       # __init__ puede acceder a __x
...                            # ya que los dos están dentro de la misma clase
>>> p = Prueba(1)
>>> p.__x   # No funciona, ya que __x es privada
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Prueba' object has no attribute '__x'
```

5.1.2.2. Accesores y mutadores

En muchas ocasiones, ocurre que necesitamos manipular el valor contenido en una variable de instancia privada, pero desde fuera del objeto.

Para ello, necesitamos definir operaciones (métodos) que nos permitan acceder y/o modificar el valor del atributos privado del objeto desde fuera del mismo.

Estos métodos pueden ser:

- **Accesores o *getters*:** permiten acceder al valor de un atributo desde fuera del objeto.
- **Mutadores o *setters*:** permiten modificar el valor de un atributo desde fuera del objeto.

Por ejemplo, si tenemos un atributo privado que deseamos manipular desde el exterior del objeto, podemos definir una pareja de métodos `get` y `set` de la siguiente forma:

```
>>> class Prueba:
...     def __init__(self, x):
...         self.set_x(x)        # En el constructor aprovechamos el setter
...
...     def get_x(self):         # Este es el getter del atributo __x
...         return self.__x
...
...     def set_x(self, x):      # Este es el setter del atributo __x
...         self.__x = x
...
```

```
>>> p = Prueba(1)
>>> p.get_x()                    # Accedemos al valor de __x
1
>>> p.set_x(5)                   # Cambiamos el valor de __x
>>> p.get_x()                    # Accedemos de nuevo al valor de __x
5
```

La pregunta es: ¿qué ganamos con todo esto?

Si necesitamos acceder y/o cambiar el valor de un atributo desde fuera del objeto, ¿por qué hacerlo privado? ¿Por qué no simplemente hacerlo público y así evitamos tener que hacer *getters* y *setters*?:

- Los atributos públicos rompen con los conceptos de *encapsulación* y de *abstracción de datos*, ya que permite acceder al interior de un objeto directamente, en lugar de hacerlo a través de operaciones.
- Como consecuencia de lo anterior, se rompe con el principio de *ocultación de información*, ya que exponemos públicamente el tipo y la representación del dato, por lo que nos resultará muy difícil cambiarlos posteriormente si en el futuro nos hace falta hacerlo.
- Además, los *setters* nos garantizan que los datos que se almacenan en un atributo cumplen con las **condiciones** necesarias.

> Las condiciones que deben cumplir en todo momento las instancias de una clase se denominan **invariantes de la clase**.

Por ejemplo: si queremos almacenar los datos de una persona y queremos garantizar que la edad no sea negativa, podemos hacer:

```
"""
Invariante: todas las personas deben tener edad no negativa.
"""
class Persona:
    def __init__(self, nombre, edad):
        self.set_nombre(nombre)
        self.set_edad(edad)

    def set_nombre(self, nombre):
        self.__nombre = nombre

    def get_nombre(self):
        return self.__nombre

    def set_edad(self, edad):
        if edad < 0:
            raise ValueError("La edad no puede ser negativa")
        self.__edad = edad

p = Persona("Manuel", 30)   # Es correcto
print(p.set_nombre())       # Imprime 'Manuel'
p.set_edad(25)              # Cambia la edad a 25
p.set_edad(-14)             # Provoca un error
p.__edad = -14              # Funcionaría si __edad no fuese privado
```

En conclusión, se recomienda:

- Hacer privados todos los miembros excepto los que sean estrictamente necesarios para poder manipular el objeto desde el exterior del mismo (su *interfaz*).
- Crear *getters* y *setters* para los atributos que se tengan que manipular desde el exterior del objeto.
- Dejar claros los invariantes de las clases en el código fuente de las mismas mediante comentarios, y comprobarlos adecuadamente donde corresponda (en los *setters*, principalmente).

6. Definiciones a nivel de clase

6.1. Variables de clase

6.2. Métodos estáticos