Programación orientada a objetos (II)

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2026/02/25 a las 20:30:00

Índice

1. Identidad e igualdad **1**
1.1. Identidad . . . 1
1.2. Igualdad . . . 5
1.2.1. `__eq__` . . . 6
1.2.2. `__hash__` . . . 7
1.3. Otros métodos mágicos . . . 11
1.3.1. `__repr__` . . . 11
1.3.2. `__str__` . . . 15

2. Encapsulación **17**
2.1. Encapsulación . . . 17
2.1.1. La encapsulación como mecanismo de agrupamiento . . . 17
2.1.2. La encapsulación como mecanismo de protección de datos . . . 18

3. Miembros de clase **31**
3.1. Variables de clase . . . 31
3.2. Métodos estáticos . . . 34

4. Clases genéricas y métodos genéricos **37**
4.1. Definición y uso . . . 37

1. Identidad e igualdad

1.1. 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 dos objetos provoca también el mismo cambio en el otro objeto.

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

Intuitivamente, se refiere al hecho de que los dos objetos sean en realidad ***el mismo* objeto**.

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

Cuando introducimos mutabilidad y estado en nuestro modelo computacional, muchos conceptos que antes eran sencillos se vuelven problemáticos.

Entre ellos, el problema de determinar si dos objetos son «el mismo objeto», es decir, si son *idénticos*.

De hecho, el problema de la identidad no se da con objetos inmutables.

Por ejemplo, supongamos que hacemos:

```
def restador(cantidad):
    def aux(otro):
        return otro - cantidad

    return aux

res1 = restador(25)
res2 = restador(25)
```

¿Son `res1` y `res2` el mismo objeto?

- Es razonable decir que sí, ya que tanto `res1` como `res2` se comportan siempre de la misma forma (las dos son funciones que restan `25` a su argumento).
- De hecho, `res1` puede sustituirse por `res2` (y viceversa) en cualquier lugar del programa sin que afecte a su funcionamiento.

Como los objetos así creados son **inmutables** y se han creado iguales, no importa demasiado si son el mismo objeto o no.

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

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

¿Son `dep1` y `dep2` el mismo objeto?

- Evidentemente no, ya que podemos obtener resultados distintos al enviarles el mismo mensaje (uno que sabemos que no cambia el estado del objeto):

  ```
  >>> dep1.retirar(20)
  80
  >>> dep1.saldo()  # Mensaje que no cambia el estado del objeto
  80
  >>> dep2.saldo()  # El mismo mensaje, da un resultado distinto en dep2
  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 podemos sustituir `dep1` por `dep2` (o viceversa) en ninguna parte del programa sin afectar a su funcionamiento.

Es otra forma de decir que, en general, **los objetos no tienen 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, se vuelve más difícil de definir de una manera formal y rigurosa qué es lo que significa que dos objetos sean «el mismo objeto».

De hecho, el significado de «el mismo» en el mundo real que estamos modelando con nuestro programa es ya bastante 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 ha cambiado de la misma forma.

Pero la única manera de saber si un objeto ha «cambiado» es observando el «mismo» objeto dos veces, en dos momentos diferentes, y comprobando si ha cambiado alguna propiedad del objeto de la primera observación a la segunda.

Por tanto, no podemos determinar si ha habido un «cambio» si no podemos determinar a priori si dos objetos son «el mismo», y no podemos determinar si son el mismo si no podemos observar los efectos de ese cambio.

Esto nos lleva a una definición circular, donde un término depende del otro y viceversa.

Por ejemplo, supongamos que Pedro y Pablo tienen un depósito con 100 € cada uno.

Si los creamos así:

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

estaremos creando dos depósitos separados e independientes.

Por tanto, las operaciones realizadas en el depósito de Pedro no afectarán al de Pablo, y viceversa.

En cambio, si los creamos así:

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

estamos definiendo a `dep_Pablo` para que sea exactamente el mismo objeto que `dep_Pedro`.

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

Estas dos situaciones, similares pero distintas, pueden provocar confusión al crear modelos computacionales.

En concreto, 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`.

Los problemas de identidad sólo se da cuando permitimos que los objetos sean mutables.

Si Pedro y Pablo sólo pudieran comprobar los saldos de sus depósitos y no pudieran realizar operaciones que cambiaran sus fondos, entonces no haría falta comprobar si los dos depósitos son realmente dos objetos separados o si por el contrario son el mismo depósito. Daría igual.

En general, siempre que no se puedan modificar los objetos, podemos suponer que un objeto compuesto se define e identifica como la suma de sus partes.

Pero esto deja de ser válido cuando incorporamos mutabilidad, porque entonces un objeto compuesto tiene una «identidad» que es algo diferente de las partes que lo componen.

Por ejemplo, un número racional viene definido por su numerador y su denominador.

El número racional $\frac{4}{3}$ está completamente determinado por su numerador 4 y su denominador 3. Es eso y nada más.

Pero no tiene sentido considerar que un número racional es un objeto mutable con identidad propia, puesto que si pudiéramos cambiar su numerador o su denominador ya no tendríamos «el mismo» número racional, sino que tendríamos otro diferente.

Si al racional $\frac{4}{3}$ le cambiamos el numerador 4 por 5, obtendríamos el nuevo racional $\frac{5}{3}$, y no tendría sentido decir que ese nuevo racional es el antiguo racional $\frac{4}{3}$ modificado. Éste ya no está.

En cambio, un depósito sigue siendo «el mismo» depósito aunque cambiemos sus fondos haciendo una retirada de efectivo.

Pero también podemos tener dos depósitos distintos con el mismo estado interno.

No olvidemos que dos objetos pueden ser **iguales** y, en cambio, no ser *idénticos*.

Esta complicación es consecuencia, no de nuestro lenguaje de programación, sino de nuestra percepción del depósito bancario como un objeto.

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) simplemente 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.

Resumiendo:

Cuando preguntamos si dos objetos son **iguales**, estamos preguntando si ***parecen* el mismo objeto**, es decir, si tienen la misma forma y el mismo contenido.

Cuando preguntamos si son **idénticos**, estamos preguntando si ***son* el mismo objeto**.

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
```

Como ya estudiamos en su día, la expresión ***o*** `is` ***p*** equivale a:

`id(`*o*`) == id(`*p*`)`

Por tanto, lo que hace es comparar el resultado de la función `id`, que devuelve un identificador único (un número) para cada objeto.

Ese identificador es la dirección de memoria donde se almacena el objeto. Por tanto, es la dirección a donde apuntan sus referencias.

1.2. Igualdad

Supongamos que queremos modelar el funcionamiento de una cola (una estructura de datos en la que los elementos entran por un lado y salen por el contrario en el orden en que se han introducido).

El código podría ser el siguiente, utilizando una lista para almacenar los elementos:

```
class Cola:
    def __init__(self, els):
        self.items = els

    def meter(self, el):
        self.items.append(el)

    def sacar(self):
        if len(self.items) == 0:
            raise ValueError("Cola vacía")
        del self.items[0]

    def elementos(self):
        return self.items
```

Si hacemos:

```
cola1 = Cola([1, 2, 3])
cola2 = Cola([1, 2, 3])
```

es evidente que `cola1` y `cola2` hacen referencia a objetos separados y, por tanto, **no son *idénticos***, ya que no se refieren *al mismo* objeto.

Aunque no son idénticos, sí podemos decir que **son *iguales*** ya que pertenecen a la misma clase, poseen el mismo estado interno (el mismo contenido) y se comportan de la misma forma al recibir la misma secuencia de mensajes en el mismo orden:

```
>>> cola1 = Cola([1, 2, 3])
>>> cola2 = Cola([1, 2, 3])
>>> cola1.meter(9)
>>> cola1.elementos()
[1, 2, 3, 9]
>>> cola1.sacar()
>>> cola1.elementos()
[2, 3, 9]
>>> cola2.meter(9)
>>> cola2.elementos()
[1, 2, 3, 9]
>>> cola2.sacar()
>>> cola2.elementos()
[2, 3, 9]
```

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

```
>>> cola1 == cola2
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 sólo si son idénticos**.

Es decir: por defecto, `x == y` sólo si `x is y`.

Esto es lo que técnicamente se denomina **igualdad por identidad**.

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

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

Podemos decir que dos colas son iguales cuando tienen el mismo estado interno. En este caso: *dos colas son iguales cuando tienen los mismos elementos en el mismo orden*.

Es lo que técnicamente se denomina **igualdad estructural**.

1.2.1. `__eq__`

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

Este método se invoca automáticamente cuando se hace una comparación con el operador `==` y el operando izquierdo es una instancia de nuestra clase. El operando derecho se envía como argumento en la llamada al método.

Dicho de otra forma:

- Si la clase de `cola1` tiene definido el método `__eq__`, entonces `cola1 == cola2` equivale a `cola1.__eq__(cola2)`.
- En caso contrario, `cola1 == cola2` seguirá valiendo lo mismo que `cola1 is cola2`, como acabamos de ver.

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

Para crear una posible implementación del método `__eq__`, podemos aprovecharnos del hecho de que dos listas son iguales cuando tienen exactamente los mismos elementos en el mismo orden (justo lo que necesitamos para nuestras colas):

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

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

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

```
>>> cola1 = Cola([9, 14, 7])
>>> cola2 = Cola([9, 14, 7])
>>> cola1 is cola2
False
>>> cola1 == cola2
True
```

```
>>> cola1.sacar()
>>> cola1 == cola2
False
>>> cola2.sacar()
>>> cola1 == cola2
True
```

1.2.2. `__hash__`

Recordemos lo que ya sabemos:

- Existen datos *hashables* y datos *no hashables*.
- Los datos *hashables* son aquellos que se pueden comparar entre sí con `==` y además llevan asociado un número entero llamado *hash*.
- Los datos *hashables* pueden guardarse en un conjunto o usarse como claves de un diccionario.
- Los datos mutables no pueden ser *hashables*.
- Si *x* es *hashable*, `hash(x)` debe devolver un número que nunca cambie durante la vida de *x*.
- Si *x* no es *hashable*, `hash(x)` lanza una excepción `TypeError`.

Lo que hace la función `hash` es llamar al método `__hash__` sobre su argumento.

Por tanto, la llamada a `hash(x)` es equivalente a hacer `x.__hash__()`.

Los métodos `__eq__` y `__hash__` están relacionados entre sí, porque siempre se tiene que cumplir la siguiente condición:

> Si *x* `==` *y*, entonces `hash(x)` debe ser igual que `hash(y)`.

Por tanto, siempre se tiene que cumplir que:

> Si *x* `==` *y*, entonces *x*`.__hash__()` `==` *y*`.__hash__()`.

Lo que equivale también a decir que:

> Si *x*`.__hash__()` `!=` *y*`.__hash__()`, entonces *x* `!=` *y*.

Para ello, debemos tener en cuenta varias consideraciones a la hora de crear nuestras clases:

1. Si una clase define su propio método `__hash__`, debe definir también un `__eq__` que vaya a juego con el `__hash__`.

   Por tanto (*contrarrecíproco* del anterior), si una clase no define su propio método `__eq__`, tampoco debe definir su propio método `__hash__`.

2. Las clases definidas por el programador ya traen de serie una implementación predefinida de `__eq__` y `__hash__` que cumplen que:

   x `==` *y* sólo si *x* **`is`** *y*, como ya vimos antes.

   x`.__hash__()` devuelve un valor que garantiza que si *x* `==` *y*, entonces `hash(`*x*`)` `==` `hash(`*y*`)`.

   (Esto se debe a que la clase *hereda* los métodos `__eq__` y `__hash__` de la clase `object`, como veremos en la siguiente unidad.)

3. Si una clase no define `__eq__` pero no se desea que sus instancias sean *hashables*, debe definir su método `__hash__` como `None` incluyendo la sentencia:

   ```
   __hash__ = None
   ```

   en la definición de la clase.

4. Si una clase define `__eq__` pero no define `__hash__`, es como si implícitamente hubiera definido `__hash__ = None` (lo hace el intérprete internamente).

   Por tanto, en ese caso sus instancias no serán *hashables*.

5. Si las instancias de la clase son mutables y ésta define `__eq__`, lo normal es que no defina `__hash__`, ya que los objetos mutables no son *hashables*, en general.

   No obstante, hay casos particulares de objetos mutables que pueden ser *hashables*, como veremos en breve.

Hemos dicho que la condición principal que se tiene que cumplir es que:

> Si *x* `==` *y*, entonces `hash(`*x*`)` `==` `hash(`*y*`)`.

Y, por tanto, se tiene que cuplir su contrarrecíproco, que es:

> Si `hash(`*x*`)` `!=` `hash(`*y*`)`, entonces *x* `!=` *y*.

Lo cual NO significa que se tenga que cumplir que:

> Si *x* `!=` *y*, entonces `hash(`*x*`)` `!=` `hash(`*y*`)`.

Sin embargo, aunque no sea necesario que se cumpla, a efectos prácticos sí que resulta conveniente cumplir la condición anterior en la medida de lo posible, ya que de esta forma ganaremos en eficiencia cuando intentemos acceder a nuestros objetos de manera directa si los almacenamos en una colección.

Por desgracia, resulta prácticamente imposible poder cumplir la condición anterior para todos los objetos, pero aún así deberíamos intentar que nuestro algoritmo de *hashing* cumpla dicha condición con el mayor número de objetos posible.

Cuando esa condición no se cumple, tenemos lo que se llama una **colisión**.

Es decir: tenemos una colisión cuando varios objetos distintos tienen el mismo valor de *hash*.

En tal caso, tenemos que:

$$x \texttt{ != } y, \text{ pero } \texttt{hash(}x\texttt{) == hash(}y\texttt{)}.$$

Como dijimos antes, las colisiones son prácticamente inevitables, pero hay que procurar implementar nuestro `__hash__` de forma que se produzcan lo menos posible, ya que mejora el rendimiento.

Dicho de otra forma, nuestro `__hash__` debe cumplir **siempre**:

$$\text{Si } x \texttt{ == } y, \text{ entonces } \texttt{hash(}x\texttt{) == hash(}y\texttt{)}$$

pero, al mismo tiempo, debe procurar cumplir **siempre que pueda**:

$$\text{Si } x \texttt{ != } y, \text{ entonces } \texttt{hash(}x\texttt{) != hash(}y\texttt{)}.$$

En realidad, una buena implementación de `__hash__` es aquella que reparte uniformemente los objetos entre los posibles valores de *hash*.

Es decir: la idea principal es que el método `__hash__` no asocie muchos objetos a un mismo valor de *hash* y al mismo tiempo haya otros valores de *hash* a los que se les asocien pocos objetos (o ninguno).

Si muchos objetos tienen el mismo *hash*, ese reparto no sería uniforme, sino que estaría muy descompensado, y provocaría un peor rendimiento en los accesos a los objetos dentro de las colecciones.

Por otra parte, el cálculo del *hash* no debería ser costoso.

Una forma sencilla de crear el `__hash__` de una clase sería usar el `hash` de una tupla que contenga las variables de estado de la clase (siempre que estas sean *hashables* también):

```
def __hash__(self):
    return hash((self.nombre, self.apellidos))
```

Las colas son mutables y, por tanto, no pueden ser *hashables*, así que no definiremos ningún método `__hash__` en la clase `Cola`.

De esta forma, como sí hemos definido un método `__eq__` en la clase, el intérprete automáticamente hará `__hash__ = None` y convertirá a las colas en *no hashables*.

Es importante no romper el contrato entre `__eq__` y `__hash__`.

Es decir, hay que garantizar que si dos objetos son iguales, sus *hash* también deben ser iguales.

De lo contrario, se pueden dar situaciones extrañas:

```
>>> import random
>>> class Rara:
...     def __hash__(self):
...         return random.randint(0, 2)
...
>>> x = Rara()
>>> s = set()
>>> s.add(x)
>>> x in s
True
>>> x in s
False
```

Aunque los objetos mutables no deberían ser *hashables*, existen dos técnicas que nos permiten obtener objetos que sean simultáneamente mutables y *hashables*:

- Implementando **igualdad estructural** por comparación entre atributos inmutables y calculando el *hash* a partir de esos atributos.
- Implementando **igualdad por identidad** y calculando el *hash* a partir de la identidad del objeto.

Si entre los atributos de un objeto hay un subconjunto de ellos que nunca cambian, se puede implementar **igualdad estructural** mediante la comparación de esos atributos.

En tal caso, el *hash* se puede calcular a partir de esos atributos que nunca cambian.

Por ejemplo, si el DNI de una persona nunca cambia, podríamos usarlo para comprobar si dos personas son iguales y además calcular su *hash*:

```
class Persona:
    def __init__(self, dni, nombre):
        self.__dni = dni
        self.__nombre = nombre

    def __eq__(self, otro):
        if type(self) != type(otro):
            return NotImplemented
        return self.__dni == otro.__dni

    def __hash__(self):
        return hash(self.__dni)

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

Así, las instancias de `Persona` serán mutables y también *hashables*.

Por otra parte, como vimos anteriormente, si una clase no implementa su propia versión de `__eq__` y no define `__hash__ = None`, entonces:

- Implementa **igualdad por identidad**, de manera que dos instancias de la clase serán iguales sólo si son idénticos (***x*** `==` ***y*** sólo si ***x*** `is` ***y***).
- Sus objetos serán *hashables*.
- El *hash* de sus instancias se calculará a partir de sus identidades.

Por ejemplo:

```
class Persona:
    def __init__(self, dni, nombre):
        self.__dni = dni
        self.__nombre = nombre

    def set_dni(self, dni):
        self.__dni = dni

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

Esta es otra forma de tener objetos mutables y *hashables*.

1.3. Otros métodos mágicos

1.3.1. `__repr__`

Existe una función llamada `repr` que devuelve la **expresión canónica** de un valor, es decir, la cadena de símbolos que mejor **representa** a ese valor.

```
>>> repr(2 + 3)
'5'
>>> 2 + 3
5
```

```
>>> repr(3.50)
'3.5'
>>> 3.5
3.5
```

La expresión que vaya en esa cadena debe ser sintáctica y semánticamente **correcta** según las reglas del lenguaje.

Además, esa expresión debe contener toda la información necesaria para **reconstruir el valor**.

El intérprete interactivo de Python usa `repr` cuando le pedimos que evalúe una expresión:

```
>>> 2 + 3  # devuelve lo mismo que repr(2 + 3) pero sin comillas
5
```

Recordemos que no todo valor tiene expresión canónica; por ejemplo, las funciones:

```
>>> repr(max)
'<built-in function max>'
```

En este caso, lo que nos devuelve `repr` no tiene la información suficiente como para construir la función `max`.

De hecho, ni siquiera es una expresión válida en el lenguaje:

```
>>> <built-in function max>
  File "<stdin>", line 1
    <built-in function max>
    ^
SyntaxError: invalid syntax
```

Al aplicar la función `repr` sobre una instancia de una clase definida por el programador, obtenemos una representación que, en general, no es correcta ni contiene la información suficiente como para representar al objeto o reconstruirlo.

Por ejemplo:

```
>>> dep = Deposito(100)
>>> dep.retirar(30)
>>> repr(dep)
'<__main__.Deposito object at 0x7fed83fd9160>'
```

Nos devuelve una cadena que que simplemente nos informa de:

- La clase a la que pertenece el objeto, que se obtiene mediante `type(dep)`.
- El `id` del objeto en hexadecimal, que se obtiene mediante `hex(id(dep))`.

Pero esa cadena no contiene ninguna expresión válida en Python y tampoco nos dice cuántos fondos contiene el depósito, por ejemplo.

Por tanto, con esa cadena no podemos reconstruir el objeto `dep`.

En este caso, lo ideal sería que `repr(dep)` devolviera una expresión **no ambigua** con la que pudiéramos **reconstruir** el objeto `dep` con toda la información que contiene (su estado interno).

Es decir, buscamos algo así:

```
>>> dep = Deposito(100)
>>> dep.retirar(30)
>>> repr(dep)
'Deposito(70)'
```

En este último caso, la cadena resultante contiene la expresión `Deposito(70)`, que sí representa adecuadamente al objeto `dep`:

```
>>> dep == Deposito(70)
True
```

Es importante no confundir la cadena `'Deposito(70)'` que devuelve `repr` con la expresión `Deposito(70)` que lleva dentro.

La función `eval` en Python evalúa la expresión contenida en una cadena y devuelve el valor resultante:

```
>>> 2 + 3 * 5
17
>>> eval('2 + 3 * 5')
```

```
17
```

Sólo se puede aplicar a cadenas:

```
>>> eval(2 + 3 * 5)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: eval() arg 1 must be a string, bytes or code object
```

Y esas cadenas tienen que ser sintáctica y semánticamente correctas:

```
>>> eval('2 + * 3 5')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    2 + * 3 5
        ^
SyntaxError: invalid syntax
```

Las funciones `eval` y `repr` están relacionadas de forma que **siempre *debería* cumplirse lo siguiente**:

`eval(repr(`*v*`)) ==` *v*

Por ejemplo:

```
>>> eval(repr(2 + 3 * 5)) == 2 + 3 * 5
True
```

En cambio, ahora mismo tenemos que:

```
>>> eval(repr(dep)) == dep
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    <__main__.Deposito object at 0x7fed83fd9160>
    ^
SyntaxError: invalid syntax
```

Lo que hace realmente la expresión `repr(v)` es llamar al método `__repr__` del objeto `v`.

Por tanto, la siguiente expresión:

```
repr(v)
```

es equivalente a ésta:

```
v.__repr__()
```

Por ejemplo:

```
>>> repr(25)
'25'
>>> (25).__repr__()
'25'
```

Lo que tenemos que hacer es definir adecuadamente un método `__repr__` en nuestra clase `Deposito`.

Ejercicio

1. Inténtalo primero.

En la clase `Deposito` podríamos hacer algo así:

```
class Deposito:
    def __repr__(self):
        return f"Deposito({self.fondos})"

    # ... resto del código
```

De esta forma, conseguimos el efecto deseado:

```
>>> dep = Deposito(100)
>>> dep.retirar(30)
>>> repr(dep)
'Deposito(70)'
>>> eval(repr(dep)) == dep
True
```

Para implementar el método `__repr__` de la clase `Persona`, podríamos probar a hacer:

```
class Persona:
    def __init__(self, dni, nombre):
        self.__dni = dni
        self.__nombre = nombre

    def __hash__(self):
        return hash(self.__dni)

    def __repr__(self):
        return f'Persona({self.__dni}, {self.__nombre})'

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

Pero obtendríamos un resultado incorrecto, porque el DNI y el nombre de la persona deberían ir entre comillas, ya que son cadenas literales:

```
>>> p = Persona('28373482X', 'Javier Rodríguez')
>>> p
Persona(28373482X, Javier Rodríguez)
>>> Persona(28373482X, Javier Rodríguez)
SyntaxError: invalid syntax
```

La solución sería aplicar la función `repr` también a los argumentos del constructor de `Persona`:

```
def __repr__(self):
    return f'Persona({repr(self.__dni)}, {repr(self.__nombre)})'
```

Esto se puede abreviar haciendo uso de la *conversión* `r` en los campos de sustitución de la *f-string*:

```
def __repr__(self):
    return f'Persona({self.__dni!r}, {self.__nombre!r})'
```

Es importante señalar que **no siempre se puede definir un `__repr__` adecuado para un objeto**.

Esto es así porque no siempre es posible representar con una expresión todo el estado interno del objeto.

De hecho, un objeto puede tener estado interno que no siquiera sea visible ni conocido en el exterior.

Por ejemplo, si cada objeto de la clase `Deposito` guardara un historial de las operaciones que se han ido realizando con ese depósito, ese historial formaría parte del estado interno del objeto pero no aparecería como parámetro en el constructor.

Por tanto, no podríamos describir con una expresión `Deposito(...)` (ni con ninguna otra) todo el estado interno del objeto.

Una forma de solucionar este problema sería hacer que el constructor de la clase pudiera recibir un parámetro adicional *opcional* que contenga ese historial:

```
class Deposito:
   def __init__(self, fondos, historial=None):
       self.fondos = fondos
       if historial is None:
           self.historial = []
       else:
           self.historial = historial

   def __repr__(self):
        return f"Deposito({self.fondos!r}, {self.historial!r})"

    # ... resto del código
```

Ese parámetro sólo se usaría en ese caso, es decir, que no estaría pensado para ser usado de forma habitual al crear objetos `Deposito`.

1.3.2. `__str__`

El método `__str__` se invoca automáticamente cuando se necesita convertir un valor al tipo `str`.

Por tanto, la siguiente expresión:

```
str(v)
```

es equivalente a ésta:

```
v.__str__()
```

Por ejemplo:

```
>>> str(25)
'25'
>>> (25).__str__()
'25'
```

Existen muchos casos donde es necesario convertir un valor a cadena, explícita o implícitamente. Por ejemplo, la función `print` convierte a cadena su argumento antes de imprimirlo.

Si la clase del objeto `v` no tiene definido el método `__str__`, por defecto se entiende que se tiene `__str__` = `__repr__`. Por tanto, en tal caso se llama en su lugar al método `__repr__`.

¿Cuál es la diferencia entre `__repr__` y `__str__`?

- La finalidad de `__repr__` es ser **no ambiguo** y deberíamos implementarlo siempre en todas nuestras clases (cuando sea posible).
- La finalidad de `__str__` es ser **legible para el usuario** y no es estrictamente necesario implementarlo.

Por ejemplo, en el módulo `datetime` tenemos clases que sirven para manipular fechas y horas.

La clase `date` del módulo `datetime` nos permite crear objetos que representan fechas:

```
>>> import datetime
>>> d = datetime.date(2020, 4, 30)
```

Al llamar a `repr` sobre `d` obtenemos una representación que nos permite reconstruir el objeto:

```
>>> repr(d)
'datetime.date(2020, 4, 30)'
```

Y al llamar a `str` sobre `d` obtenemos una versión más fácil de entender para un ser humano:

```
>>> str(d)
'2020-04-30'
```

Se puede observar aquí que el intérprete usa `repr` para mostrar la forma normal del objeto:

```
>>> d
'datetime.date(2020, 4, 30)'
```

Y que `print` usa `str` para imprimir el objeto de una forma legible:

```
>>> print(d)
2020-04-30
```

Recordemos que `print` imprime una cadena por la salida (sin comillas) y devuelve `None`.

2. Encapsulación

2.1. 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.

2.1.1. 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 o **ligado** a un identificador.

En definitiva, un ciudadano de primera clase es **un valor de un determinado tipo**, simple o compuesto.

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.

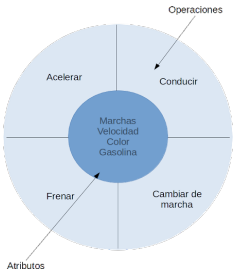
2.1.2. La encapsulación como mecanismo de protección de datos

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.

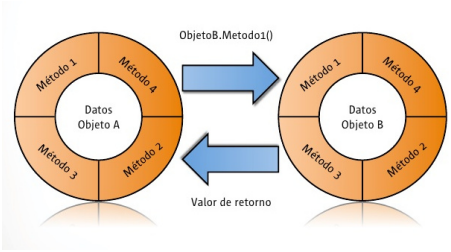
Según esto, podemos imaginar que:

- 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 esos 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 son las que aparecen en la **especificación** de su tipo abstracto y, por tanto, definen de qué manera podemos manipular al objeto desde el exterior del mismo.



Las operaciones forman una *cápsula*

2.1.2.1. Visibilidad

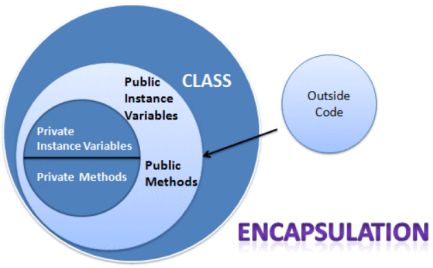
Para garantizar esta restricción de acceso, los lenguajes de programación a menudo facilitan un mecanismo por el cual el programador puede definir la **visibilidad** de cada miembro de una clase.

De esta forma, el programador puede «marcar» que determinadas variables de instancia 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 tiene visibilidad privada, sólo podrá accederse a él desde dentro de esa clase, pero no desde fuera de ella.
- **Visibilidad *pública*:** si un miembro 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 *públicos* de ese objeto**.



Miembros *públicos* y *privados*

Cada lenguaje de programación tiene su propia manera de implementar mecanismos de visibilidad.

En Python, el mecanismo es muy sencillo:

> Si el nombre de un miembro (de clase o de objeto) definido dentro del cuerpo de una clase 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 __
```

El método `__uno` se ha creado dentro de la clase `Prueba` y pertenece a la misma clase `Prueba`, así que ese método es *privado*, se puede acceder a él dentro del cuerpo de la clase `Prueba`, pero no es visible fuera de ella.

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'
```

La variable de instancia `__x` se ha creado dentro de la clase `Prueba` y pertenece a un objeto de la misma clase `Prueba`, así que esa variable es *privada*, se puede acceder a ella dentro del cuerpo de la clase `Prueba`, pero no es visible fuera de ella.

2.1.2.2. 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 de la variable de instancia privada del objeto desde fuera del mismo.

Estos métodos pueden ser:

- **Accesores o *getters*:** permiten acceder al valor de una variable de instancia privada desde fuera del objeto.
- **Mutadores o *setters*:** permiten modificar el valor de una variable de instancia privada desde fuera del objeto.

Por ejemplo, si tenemos una variable de instancia privada 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 de la variable __x
...         return self.__x
...
...     def set_x(self, x):      # Este es el setter de la variable __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?

2.1.2.3. Propiedades

En Python, una **propiedad** (*property*) es una forma elegante de definir atributos con *getters* y *setters* sin perder la sintaxis de acceso usando el operador punto (`.`).

Son útiles cuando:

- Quieres validar o transformar el valor al asignarlo.
- Necesitas que un atributo sea de solo lectura.
- Quieres exponer un atributo «calculado» como si fuera normal.
- Tienes un atributo pero necesitas añadir control.
- Quieres que un atributo calculado se acceda como uno almacenado.
- No quieres romper la *interfaz de la clase* (un concepto que veremos a continuación) si después necesitas validación.

Definición básica con `@property`:

```
class Persona:
    def __init__(self, edad):
        self.__edad = edad   # atributo privado

    @property
    def edad(self):
        return self.__edad   # getter

    @edad.setter
    def edad(self, valor):
        if valor < 0:
            raise ValueError("La edad no puede ser negativa")
        self.__edad = valor  # setter
```

Ejemplo de uso:

```
p = Persona(30)
print(p.edad)   # llama al getter → 30

p.edad = 40     # llama al setter
p.edad = -1     # ValueError
```

Usamos `__edad` como atributo privado para evitar recursión y señalar que no debe tocarse desde fuera (*encapsulación*).

Propiedad de sólo lectura:

```
from math import pi

class Circulo:
    def __init__(self, radio):
        self.radio = radio

    @property
    def area(self):
        return pi * self.radio ** 2
```

Ejemplo de uso:

```
c = Circulo(3)
print(c.area)   # OK
c.area = 5      # Error: no tiene setter
```

Usar `property()` como función en lugar de usar decoradores:

```
class Persona:
    def __init__(self, edad):
        self.__edad = edad

    def get_edad(self):
        return self.__edad

    def set_edad(self, valor):
        if valor < 0:
            raise ValueError("Edad negativa")
        self.__edad = valor

    edad = property(get_edad, set_edad)
```

2.1.2.4. Invariantes de clase

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

- Las variables de instancia públicas del objeto 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.
- Ya sabemos que con eso 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.
- Pero además, los *setters* nos garantizan que los valores que se almacenan en una variable de instancia 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:

```python
class Persona:
    """Invariante: todas las personas deben tener edad no negativa."""
    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 privada
```

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).

El concepto de invariante de una clase, aunque puede parecer nuevo, en realidad es el mismo concepto que ya vimos al estudiar las **abstracciones de datos**.

Entonces dijimos que una abstracción de datos se define por unas **operaciones** y por las **propiedades** que deben cumplir esas operaciones.

También dijimos que esas propiedades se describen como *ecuaciones* en la **especificación** del tipo abstracto (y, por tanto, se deben cumplir independientemente de la implementación).

Cuando implementamos un tipo abstracto mediante una clase, **algunas de esas propiedades se traducen en invariantes** de la clase.

En cambio, otras de esas propiedades **no serán *invariantes* de la clase, sino *condiciones* que tienen que cumplir los métodos** (es decir, las operaciones) al entrar o salir de los mismos.

Esas condiciones son las que forman la **especificación funcional** de cada método de la clase.

Recordemos que una especificación funcional contiene dos condiciones:

- **Precondición**: condición que tiene que cumplir el método para poder ejecutarse.

- **Postcondición**: condición que tiene que cumplir el método al acabar de ejecutarse.

Si se cumple la precondición de un método y éste se ejecuta, al finalizar su ejecución se debe cumplir su postcondición.

Forman una especificación porque describen qué tiene que hacer el método sin entrar a ver el cómo.

Resumiendo:

- Las clases implementan **tipos abstractos de datos**.
- Los tipos abstractos de datos se **especifican** indicando sus **operaciones** y las **propiedades** que deben cumplir esas operaciones.
- Esas propiedades se traducirán en:
  - **Invariantes** de la clase.
  - **Precondiciones** o **postcondiciones** de los métodos que implementan las operaciones del tipo abstracto.
- El **usuario de la clase** es **responsable** de garantizar que se cumple la **precondición** de un método cuando lo invoca.
- La **implementación** de la clase es **responsable** de garantizar que se cumple en todo momento las **invariantes** de la clase, así como las **postcondiciones** de los métodos cuando se les invoca en un estado que cumple su precondición.

2.1.2.5. Interfaz y especificación de una clase

La **interfaz de una clase** (o de un objeto de esa clase) está formada por todo lo que es público y visible desde fuera de la clase.

En concreto, la interfaz de una clase indica:

- El **nombre de la clase**.
- El nombre y tipo de las **variables de instancia públicas**.
- La **signatura** de los **métodos públicos**.

Es un concepto puramente sintáctico, porque describe **qué** proporciona la clase pero no qué propiedades debe cumplir (para qué sirve la clase).

Por tanto, podemos decir que el usuario de la clase no tiene suficiente con conocer la interfaz de la clase.

También necesita saber qué hace, y eso no se indica en la interfaz.

La **especificación de una clase** representa todo lo que es necesario conocer para usar la clase (y, por tanto, cualquier objeto de esa clase).

Describe qué hace la clase (o el objeto) sin detallar cómo.

Tiene un papel similar a la especificación de un tipo abstracto de datos.

Está formado por:

- La **interfaz** de la clase.

- Los **invariantes** de la clase.
- La **especificación funcional** (precondición, postcondición y signatura) de todos los **métodos públicos** de la clase.
- **Documentación adicional** que explique la función de la clase y sus operaciones, así como posible información extra que pueda resultar de interés para el usuario de la clase.

2.1.2.6. Asertos

La **comprobación** continua de las condiciones (invariantes, precondiciones o postcondiciones) cada vez que se actualiza el estado interno de un objeto puede dar lugar a **problemas de rendimiento**, ya que las comprobaciones consumen tiempo de CPU.

Una técnica alternativa a la comprobación con sentencias condicionales (`if`s) es el uso de **asertos**.

Un aserto es una **condición que se debe cumplir en un determinado punto del programa**, de forma que el programa abortará en ese punto si no se cumple.

Para insertar un aserto en un punto del programa, se usa la sentencia `assert`.

El código anterior quedaría así usando `assert`:

```
"""
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):
        assert edad >= 0      # La edad tiene que ser >= 0
        self.__edad = edad
```

El intérprete comprobará el aserto cuando el flujo de control llegue a la sentencia `assert` y, en caso de que no se cumpla, lanzará una excepción de tipo `AssertionError`.

Lo interesante de los asertos es que podemos pedirle al intérprete que ignore todas las sentencias `assert` cuando ejecute el código.

Para ello, usamos la opción `-O` al llamar al intérprete de Python desde la línea de comandos del sistema operativo:

```
# prueba.py
print("Antes")
assert 1 == 0
print("Después")
```

```
$ python prueba.py
Antes
Traceback (most recent call last):
  File "prueba.py", line 2, in <module>
    assert False
AssertionError
$ python -O prueba.py
Antes
Después
```

Con la opción `-O` (de «**O**ptimizado») podemos elegir entre mayor rendimiento o mayor seguridad al ejecutar nuestros programas.

Aún así, no siempre es conveniente poder saltarse los asertos. De hecho, a veces lo mejor sigue siendo comprobar condiciones con un **`if`** y lanzar un error adecuado si la condición no se cumple.

Por ejemplo, si intentamos retirar fondos de un depósito pero no hay saldo suficiente, eso se debería comprobar siempre:

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

    def retirar(self, cantidad):
        if cantidad > self.fondos:                    # Si no hay fondos:
            raise ValueError("Fondos insuficientes")  # Error
        self.fondos -= cantidad
        return self.fondos

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

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

¿Cuándo usar asertos y cuándo usar excepciones?

- Las **excepciones** se deben usan para detectar errores de programación de los **usuarios** del método o clase.
- Los **asertos** se deben usar para detectar errores de funcionamiento del **interior** del método o clase, es decir, errores que haya cometido el **implementador** o **creador** del método o clase.
- Además, los asertos están pensados para detectar errores muy anormales que no se espera que se puedan capturar ni gestionar.
- Por eso, el objetivo principal de los asertos es servir al creador de una clase como mecanismo de comprobación de que su clase funciona correctamente.

Resumiendo:

- Un **invariante de una clase** es una condición que se debe cumplir durante toda la vida de todas las instancias de una clase.
- Una **precondición de un método** es una condición que se debe cumplir justo antes de ejecutar el método.

- Una **postcondición de un método** es una condición que se debe cumplir justo al finalizar la ejecución del método.
- Un **aserto** es una condición que se debe cumplir en un determinado punto del programa.
- Para **implementar invariantes de clase, precondiciones o postcondiciones de métodos** se pueden usar excepciones, asertos y sentencias `assert` en puntos adecuados del código fuente de la clase.

¡CUIDADO!

Supogamos que tenemos el siguiente código que implementa colas:

```
class Cola:
    """Invariante: self.__cantidad == len(self.__items)."""
    def __init__(self):
        self.__cantidad = 0
        self.__items = []

    def meter(self, el):
        self.__items.append(el)
        self.__cantidad += 1

    def sacar(self):
        if self.__cantidad == 0:
            raise ValueError("Cola vacía")
        del self.__items[0]
        self.__cantidad -= 1

    def get_items(self):
        return self.__items
```

Se supone que la variable de instancia `__items` es privada y, por tanto, sólo se puede acceder a ella desde el interior de la clase.

El método `get_items` es un *getter* para la variable de instancia `__items`.

En teoría, los únicos métodos con los que podemos modificar el contenido de la variable de instancia `__items` son `meter` y `sacar`.

Sin embargo, podemos hacer así:

```
c = Cola()
c.meter(1)
c.meter(2)
l = c.get_items() # Obtenemos la lista contenida en __items
del l[0]          # Eliminamos un elemento de la lista desde fuera de la cola
```

Esto se debe a que `get_items` devuelve una referencia a la lista contenida dentro de la instancia de `Cola`, con lo cual podemos modificar la lista desde el exterior sin necesidad de usar los *setters*.

Por tanto, podemos romper los invariantes de la clase, ya que ahora se cumple que `c.__cantidad` vale 2 y `len(c.__items)` vale `1` (no coinciden).

Para solucionar el problema, tenemos dos opciones:

- Quitar el método `get_items` si es posible.

- Si es estrictamente necesario que exista, cambiarlo para que no devuelva una referencia a la lista, sino una **copia** de la lista:

```
def get_items(self):
    return self.__items[:]
```

2.1.2.7. Un ejemplo completo

Recordemos la especificación del tipo *pila* **inmutable**:

spec *pila*
  parámetros
    elemento
  operaciones
    `pvacia` : $\longrightarrow$ *pila*
    `apilar` : *pila* $\times$ *elemento* $\longrightarrow$ *pila*
    parcial `desapilar` : *pila* $\longrightarrow$ *pila*
    parcial `cima` : *pila* $\longrightarrow$ *elemento*
    `vacia?` : *pila* $\longrightarrow \mathbb{B}$
  var
    p : *pila*; x : *elemento*
  ecuaciones
    `cima`(`apilar`(p, x)) $\equiv x$
    `desapilar`(`apilar`(p, x)) $\equiv p$
    `vacia?`(`pvacia`) $\equiv V$
    `vacia?`(`apilar`(p, x)) $\equiv F$
    `cima`(`pvacia`) $\equiv$ **error**
    `desapilar`(`pvacia`) $\equiv$ **error**

La especificación del mismo tipo *pila* pero **mutable** podría ser:

spec *pila*
  parámetros
    elemento
  operaciones
    `pila` : $\longrightarrow$ *pila*
    `apilar` : *pila* $\times$ *elemento* $\longrightarrow \emptyset$
    parcial `desapilar` : *pila* $\longrightarrow \emptyset$
    parcial `cima` : *pila* $\longrightarrow$ *elemento*
    `vacía?` : *pila* $\longrightarrow \mathbb{B}$
    _ `==` _ : *pila* $\times$ *pila* $\longrightarrow \mathbb{B}$
  var
    p, p_1, p_2 : *pila*; x : *elemento*
  ecuaciones
    p_1 `==` $p_2 \equiv$ «p_1 y p_2 tienen los mismos elementos en el mismo orden»
    `vacía?`(p) $\equiv p$ `==` `pila`
    `apilar`(p, x) { Apila el elemento x en la cima de la pila p }
    `desapilar`(p) { Saca de la pila p el elemento situado en su cima }
    `cima`(p) $\equiv$ «el último elemento apilado en p y aún no desapilado»
    `vacía?`(p) $\Longrightarrow$ `desapilar`(p) $\equiv$ **error**

`vacía?`(p) $\Longrightarrow$ `cima`(p) $\equiv$ **error**

A veces, la especificación de un tipo abstracto resulta más conveniente redactarla en lenguaje natural, simplemente porque queda más fácil de entender o más claro o fácil de leer.

Por ejemplo, podríamos crear un documento de **especificación en lenguaje natural** del tipo abstracto *pila* explicando qué funcionalidad tiene y las operaciones que contiene:

Tipo: *pila*

Define una pila de elementos, de forma que se van almacenando en el orden en que han sido introducidos y se van extrayendo en orden contrario siguiendo una estrategia *LIFO* (*L*ast *I*n, *F*irst *O*ut).

Los elementos pueden ser de cualquier tipo.

Dos pilas son iguales si tienen los mismos elementos y en el mismo orden.

Operaciones constructoras y modificadoras:

`pila()` $\longrightarrow$ *pila*

Crea una pila vacía (es decir, sin elementos) y la devuelve.

`apilar(`*p*: *pila*, *elem*`)` $\longrightarrow \emptyset$

Introduce el elemento *elem* encima de la pila *p*. Ese elemento pasa a estar ahora en la cima de la pila, por lo que tras su ejecución se debe cumplir que `cima(`*p*`) ==` *elem*. La operación no devuelve ningún resultado.

`desapilar(`*p*: *pila*`)` $\longrightarrow \emptyset$

Extrae de la pila *p* el elemento situado en la cima. Si *p* está vacía, da error. El elemento que queda ahora en la cima es el que había justo antes de apilar el elemento recién extraído. La operación no devuelve ningún resultado.

Operaciones selectoras:

p_1: *pila* `==` p_2: *pila* $\longrightarrow \mathbb{B}$

Devuelve *V* si p_1 y p_2 son dos pilas iguales, y *F* en caso contrario.

Dos pilas son iguales si tienen los mismos elementos y en el mismo orden.

`vacía?(`*p*: *pila*`)` $\longrightarrow \mathbb{B}$

Devuelve *V* si la pila *p* no tiene elementos, y *F* en caso contrario.

`cima(`*p*: *pila*`)` $\longrightarrow$ *cualquiera*

Devuelve el elemento situado en la cima de la pila. Si la pila está vacía, da error.

El tipo del dato devuelto es el tipo del elemento que hay en la cima.

Una posible implementación con una clase Python podría ser:

```python
class Pila:
    def __init__(self):
        self.__elems = []

    def __eq__(self, otra):
        if type(self) != type(otra):
            return NotImplemented
        return self.__elems == otra.__elems

    def vacia(self):
        return self.__elems == []

    def apilar(self, elem):
        self.__elems.append(elem)

    def desapilar(self):
        if self.vacia():
            raise ValueError('No se puede desapilar una pila vacía')
        self.__elems.pop()

    def cima(self):
        if self.vacia():
            raise ValueError('Una pila vacía no tiene cima')
        return self.__elems[-1]
```

Resulta curioso observar que la implementación, en este caso, es probablemente más corta, elegante, precisa y fácil de entender que cualquiera de las especificaciones que hemos visto anteriormente.

De hecho, si considerásemos al lenguaje Python como un lenguaje con el que escribir especificaciones, el código anterior resultaría la mejor especificación de todas las que hemos visto.

Eso se debe a que la riqueza de tipos de Python, junto con su sintaxis sencilla, lo hacen un lenguaje fácil de leer y con el que se pueden expresar muchas ideas con pocos caracteres.

Así que una implementación puede verse como una especificación, y un lenguaje de programación puede usarse para escribir especificaciones (combinándolo, posiblemente, con algo de lenguaje natural).

Aunque esto puede parecer raro en un principio, es algo que se hace a menudo.

Las especificaciones escritas con un lenguaje de programación se denominan **especificaciones operacionales**.

3. Miembros de clase

3.1. Variables de clase

Supogamos que el banco que guarda los depósitos paga intereses a sus clientes en un porcentaje fijo sobre el saldo de sus depósitos.

Ese porcentaje puede cambiar con el tiempo, pero es el mismo para todos los depósitos.

Como es un valor compartido por todos los objetos de la misma clase, se guardará en una variable local a la clase y, por tanto, se almacenará como un atributo de la propia clase, no en una instancia concreta de la clase.

Esas variables que pertenecen a la propia clase (en lugar de a instancias concretas) se denominan **variables de clase** o **variables estáticas**, a diferencia de las **variables de instancia** que hemos estado usando hasta ahora y que pertenecen a las instancias de la clase.

Las variables de clase se pueden crear y modificar mediante **sentencias de asignación** directamente en el *cuerpo* de la clase, fuera de cualquier definición de método:

```
class Deposito:
    interes = 0.02  # Una variable de clase

    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
```

Fuera de la clase, o dentro de un método de la clase, estas variables también se pueden crear y manipular a través de una referencia a la clase usando la sintaxis:

clase . *atributo*

```
>>> Deposito.interes
0.02
>>> Deposito.interes = 0.08
>>> Deposito.interes
0.08
>>> Deposito.nueva = 5
>>> Deposito.nueva
5
```

Esto nos indica que las variables de clase se almacenan en la propia clase, es decir, en el objeto que representa a la clase.

Las variables de clase también se pueden acceder como cualquier variable de instancia, a partir de una instancia de la clase:

```
>>> d1 = Deposito(100)
>>> d2 = Deposito(400)
>>> Deposito.interes          # Accede al interés de la clase Deposito
0.02
>>> d1.interes               # También
0.02
>>> d2.interes               # También
0.02
>>> Deposito.interes = 0.08   # Cambia la variable de clase
>>> Deposito.interes
0.08                         # Se comprueba que ha cambiado
```

```
>>> d1.interes                # Cambia también para la instancia
0.08
>>> d2.interes                # Cambia para todas las instancias
0.08
```

Pero esta segunda forma no es conveniente, como ahora veremos.

Si intentamos cambiar el valor de una variable de clase desde una instancia, lo que ocurre en realidad es que **creamos una nueva variable de instancia con el mismo nombre** que la variable de clase:

```
>>> Deposito.interes
0.02
>>> d1 = Deposito(100)
>>> d1.interes
0.02
>>> d1.interes = 0.08          # Crea una nueva variable de instancia
>>> d1.interes                 # Accede a la variable de instancia
0.08
>>> Deposito.interes           # Accede a la variable de clase
0.02
```

Esto ocurre porque la variable de instancia se almacena en el objeto, no en la clase, y al acceder desde el objeto tiene preferencia.

Por ello, es conveniente acostumbrarse a usar siempre el nombre de la clase para acceder y cambiar el valor de una variable de clase, en lugar de hacerlo a través de una instancia.

Para acceder al valor de una variable de clase dentro de un método, aunque sea de la misma clase, usaremos la misma sintaxis *clase*`.`*variable*, ya que de lo contrario la variable no estará en el entorno:

```
class Deposito:
    interes = 0.02   # Una variable de clase

    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

    def total(self):
        # Accede a la variable de clase Deposito.interes para calcular
        # el saldo total más los intereses (no funciona si intentamos
        # poner interes en lugar de Deposito.interes):
        return self.saldo() * (1 + Deposito.interes)
```

Ejercicios

2. ¿Qué ocurre si en el método `total` del código anterior usamos `interes` en lugar de `Deposito.interes`? ¿Por qué?
3. ¿Qué problema puede haber si en el método `total` del código anterior usamos `self.interes` en lugar de `Deposito.interes`? ¿Por qué?

3.2. Métodos estáticos

Los **métodos estáticos** son métodos definidos dentro de una clase pero que **no se ejecutan sobre ninguna instancia**.

Al no haber instancia, **los métodos estáticos no reciben ninguna instancia como argumento** a través del primer parámetro `self`.

En realidad, un método estático es básicamente **una función normal definida dentro de una clase** y que está pensada para ser ejecutada como cualquier otra función.

Por contraste, los métodos que se ejecutan sobre un objeto se denominan **métodos de instancia**, para distinguirlos de los estáticos.

Al estar definida dentro de la clase, para acceder a un método estático desde fuera de la clase o desde un método de la propia clase, hay que usar el operador punto (`.`) desde una referencia a la clase.

Por ejemplo, supongamos una clase `Numero` que representa números.

Una manera de implementarla sin métodos estáticos sería suponer que cada instancia de la clase representa un número y que las operaciones modifican ese número, recibiendo el resto de operandos mediante argumentos:

```
class Numero:
    def __init__(self, valor):
        self.set_valor(valor)

    def set_valor(self, valor):
        self.__valor = valor

    def get_valor(self):
        return self.__valor

    def suma(self, otro):
        self.set_valor(self.get_valor() + otro)

    def mult(self, otro):
        self.set_valor(self.get_valor() * otro)

n = Numero(4)
n.suma(7)
print(n.get_valor())       # Imprime 11
n.mult(5)
print(n.get_valor())       # Imprime 55
```

Para crear un método estático dentro de una clase:

- Se añade el **decorador** `@staticmethod` justo encima de la definición del método.
- El método no debe recibir el parámetro `self`.

Sabiendo eso, podemos crear una clase `Calculadora` que ni siquiera haría falta instanciar y que contendría las operaciones a realizar con los números.

Esas operaciones serían métodos estáticos.

Al estar definidos dentro de la clase `Calculadora`, para acceder a ellos habrá que usar el operador punto (`.`).

Tendríamos, por tanto:

```
class Calculadora:
    @staticmethod
    def suma(x, y):
        return x + y

    @staticmethod
    def mult(x, y):
        return x * y

s = Calculadora.suma(4, 7)   # Llamamos al método suma directamente sobre la clase
print(s)                     # Imprime 11
m = Calculadora.mult(11, 5)  # Llamamos al método mult directamente sobre la clase
print(m)                     # Imprime 55
```

De este modo, los números no se modifican.

Lo que hace básicamente el decorador `@staticmethod` es decirle al intérprete que se salte el mecanismo interno habitual de pasar automáticamente una referencia del objeto como primer parámetro del método (el que normalmente se llama `self`).

Por ejemplo, con la clase `Numero`, si tenemos que:

```
n = Numero(4)
```

es lo mismo hacer:

```
n.suma(5)
```

que hacer:

```
Numero.suma(n, 5)
```

ya que `suma` es un método de instancia en la clase `Numero`.

(Esta última forma no se usa nunca, ya que confunde al lector.)

En cambio, en la clase `Calculadora`, el método `suma` es estático, no hay objeto *sobre* el que actuar, así que no se pasa automáticamente ninguna referencia.

Todos los argumentos deben pasarse expresamente al método:

```
s = Calculadora.suma(4, 3)
```

Como lo que se reciben son enteros y no instancias de `Numero`, no los puede modificar.

Podemos combinar métodos estáticos y no estáticos en la misma clase.

En tal caso, debemos recordar que los métodos estáticos de una clase no pueden acceder a los miembros no estáticos de esa clase, ya que no disponen de la referencia al objeto (`self`).

En cambio, un método estático sí puede acceder a variables de clase o a otros métodos estáticos (de la misma clase o de cualquier otra clase) usando el operador punto (`.`).

Ejemplo

```
class Numero:
    def __init__(self, valor):
        self.set_valor(valor)

    def set_valor(self, valor):
        self.__valor = valor

    def get_valor(self):
        return self.__valor

    def suma(self, otro):
        self.set_valor(self.get_valor() + otro)

    def mult(self, otro):
        self.set_valor(self.get_valor() * otro)

    @staticmethod
    def suma_es(x, y):
        return x + y

    @staticmethod
    def mult_es(x, y):
        ret = 0
        for i in range(y):
            # Hay que poner «Numero.»:
            ret = Numero.suma_es(ret, x)
        return ret
    return ret
```

```
# El número es 4:
n = Numero(4)
# Ahora el número es 9:
n.suma(5)
# Devuelve 15:
s = Numero.suma_es(7, 8)
# Devuelve 56:
m = Numero.mult_es(7, 8)
```

4. Clases genéricas y métodos genéricos

4.1. Definición y uso

Al igual que existen *funciones genéricas*, también existen clases genéricas y métodos genéricos.

Esquemáticamente, las **clases genéricas** tienen la siguiente forma:

```
class Pila[T]: ...
```

Aquí, `T` es una **variable de tipo** que representa a un tipo cualquiera.

Al usar la sintaxis `[T]`, expresamos el hecho de que `T` representa un **parámetro de tipo** para la clase, y sirve para expresar el hecho de que la clase que estamos definiendo es genérica.

Las clases genéricas pueden contener *métodos genéricos* y *métodos no genéricos*.

Los **métodos genéricos** son métodos en cuya definición aparecen, como variables de tipo, algunos o todos los parámetros de tipo de la clase en la que se está definiendo el método.

Esas variables de tipo no se escriben en la definición del método usando la sintaxis `[T]`, ya que no son parámetros de tipo del método, sino de la clase.

Los métodos genéricos pueden tener, además, sus propios parámetros de tipo que sí aparecerían entre corchetes.

En definitiva, las clases genéricas y los métodos genéricos van de la mano y se definen de forma coordinada.

Un ejemplo de pila genérica podría ser la siguiente:

```
class Pila[T]:
    """
    Una pila de elementos de tipo T.
    """

    def __init__(self) -> None:
        self.__items: list[T] = []

    def apilar(self, item: T) -> None:
        """Añade un elemento a la pila."""
        self.__items.append(item)

    def desapilar(self) -> T | None:
        """Saca y devuelve el último elemento de la pila."""
        if self.esta_vacia():
            return None
        return self.__items.pop()

    def cima(self) -> T | None:
        """Devuelve el elemento en la cima sin quitarlo."""
        if self.esta_vacia():
            return None
        return self.__items[-1]

    def esta_vacia(self) -> bool:
```

```
        """Indica si la pila está vacía."""
        return len(self.__items) == 0
```

Ejemplo de uso:

```
# Pila de enteros
pila_enteros = Pila[int]()
pila_enteros.apilar(10)
pila_enteros.apilar(20)
print(pila_enteros.cima())      # 20
print(pila_enteros.desapilar()) # 20
print(pila_enteros.desapilar()) # 10
print(pila_enteros.desapilar()) # None

# Pila de cadenas
pila_cadenas = Pila[str]()
pila_cadenas.apilar("hola")
pila_cadenas.apilar("mundo")
print(pila_cadenas.cima())      # "mundo"
```

Ventajas de hacer que la clase sea genérica:

- Puedes crear `Pila[int]`, `Pila[str]`, `Pila[float]`, etc.
- El verificador de tipos (`mypy`, `pyright`) sabrá qué tipo de dato se espera en cada instancia y marcará errores si se intenta usar otro.

Bibliografía

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. MIT Press ; McGraw-Hill.

DeNero, John. n.d. *Composing Programs*. http://www.composingprograms.com.

Python Software Foundation. n.d. *Sitio Web de Documentación de Python*. https://docs.python.org/3.