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** (*POO*).

> **Definición:**
>
> La **programación orientada a objetos** es un paradigma de programación en el que los programas se ven como formados por entidades llamadas **objetos** que recuerdan su propio **estado interno**

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.

Ejercicio

1. Investiga en Wikipedia los principales lenguajes orientados a objetos que existen en el mercado. ¿En qué año salieron? ¿Cuál influyó en cuál? ¿Cuáles son los más usados a día de hoy?

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 define 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⟩+
```

Las clases definen un **espacio de nombres**, y todas las definiciones que se hagan dentro de la clase se almacenarán en su espacio de nombres.

Al definir su propio ámbito, todos los elementos definidos dentro de la clase son **locales** a la clase.

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

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

El cuerpo de un método define un nuevo ámbito, pero ese ámbito no está incluido dentro del ámbito de la clase.

Por tanto, desde el interior de un método no se puede acceder directamente al resto de miembros de la clase, ya que están definidos en un ámbito distinto que no es accesible desde el método.

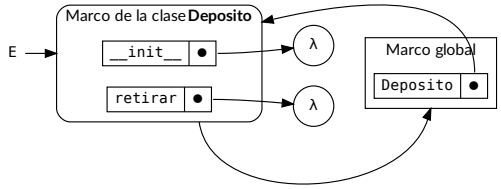
Para poder acceder a ellos habrá que usar usar el operador punto (`.`), con el que accederemos al interior del espacio de nombres de la clase.

Por ejemplo, en el código anterior:

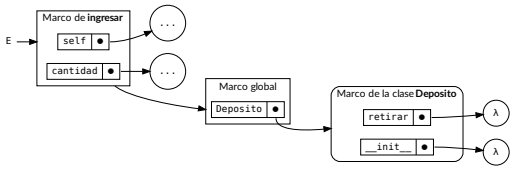
```
1  class Deposito:
2      def __init__(self, fondos):
3          self.fondos = fondos
4
5      def retirar(self, cantidad):
6          if cantidad > self.fondos:
7              return 'Fondos insuficientes'
8          self.fondos -= cantidad
9          return self.fondos
10
11     def ingresar(self, cantidad):
12         self.fondos += cantidad
13         return self.fondos
14
```

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

En la línea 10 tendríamos el siguiente entorno:



Y en la línea 12, el entorno sería:



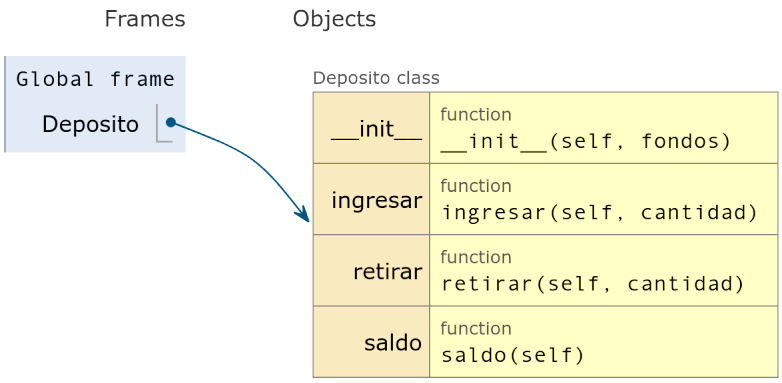
Con las **clases** ocurre exactamente igual que con los **módulos**:

- Al entrar en el ámbito de la clase, se crea un nuevo marco en memoria que contiene las definiciones que se van creando durante la ejecución de la clase.
- **Ese marco no desaparece al salir del ámbito**, sino que permanece en memoria formando una estructura tipo diccionario que almacena el **espacio de nombres** de la clase y que representa a la clase dentro de la memoria.
- La clase acaba siendo un dato más, almacenado en el montículo, que contiene su espacio de nombres, y que se ligará al nombre de la clase en el ámbito donde se haya definido la clase.
- El nombre de la clase representa el identificador de una variable que almacena una referencia a ese dato, lo que nos permite acceder a la clase desde fuera usando el operador punto (`.`).

Ese «dato» clase permanecerá en memoria mientras exista, al menos, una referencia que apunte a él.

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

Esa estructura en forma de diccionario representa a la clase dentro de la memoria («*es*» la clase), y 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

Ese diccionario representa a la clase en memoria, y nos indica que las clases son **espacios de nombres**.

Recordemos que los espacios de nombres son estructuras que almacenan correspondencias entre un nombre y un valor (como puede ser una función o un método, como es el caso aquí).

Otros espacios de nombres que hemos visto hasta ahora en el curso son los **marcos** o los módulos.

Como las clases son espacios de nombres, debemos usar el operador punto (`.`) para acceder al contenido de una clase, indicando la referencia a la clase y el nombre del contenido al que se desea acceder:

```
>>> Deposito.retirar
<function Deposito.retirar at 0x7f6f04885160>
```

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.

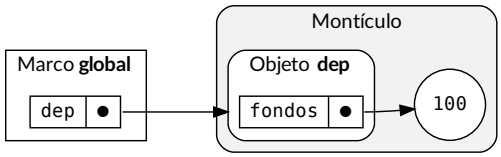
2.3.1. Atributos

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`

Los objetos son **espacios de nombres**, ya que cada objeto almacena las correspondencias entre los atributos del objeto y sus valores.

Gracias a ello, podemos usar el operador punto (`.`) para acceder a un atributo del objeto usando la sintaxis:

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
```

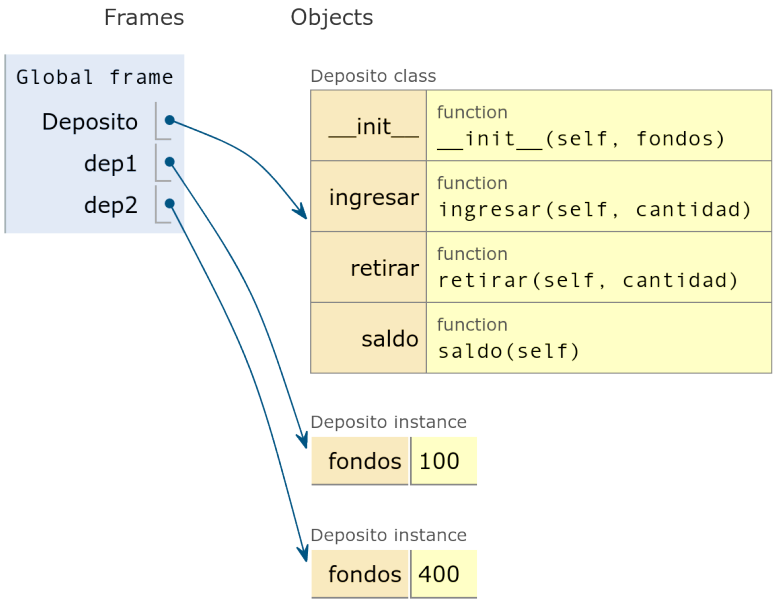
Y podemos cambiar el valor del atributo mediante asignación (cosa que, en general, no resultaría aconsejable):

```
>>> 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

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.

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:

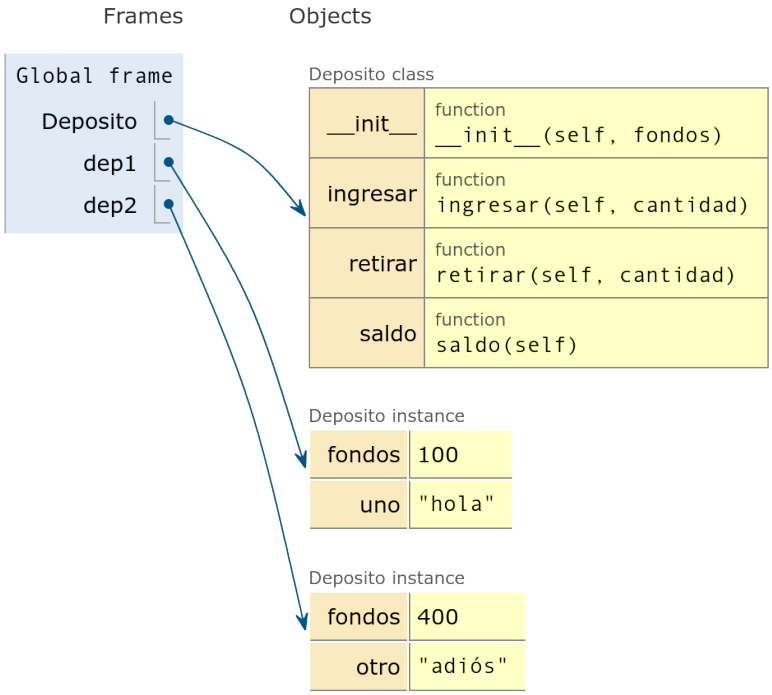
```
class Deposito:
    def __init__(self, fondos):
        self.fondos = fondos

    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:

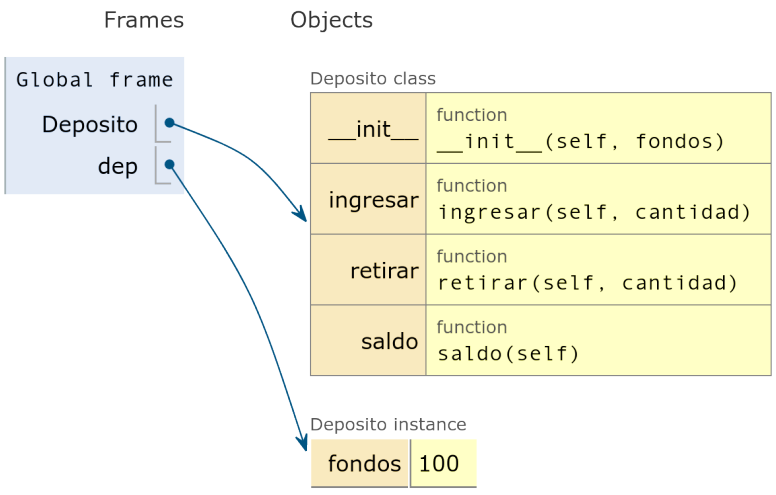
```
class Deposito:
    def __init__(self, fondos):
        self.fondos = fondos

    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:

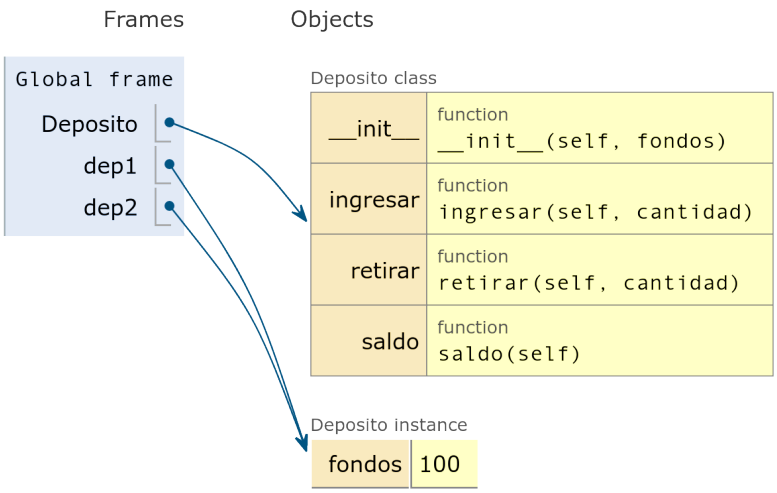
```
dep1 = Deposito(100)
dep2 = dep1
```

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).

No olvidemos que quien almacena los métodos es la clase, no el objeto.

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.

Para la clase `Despacho`, `retirar` es una función, mientras que para el objeto `dep` es un método.

Son cosas distintas, y el intérprete los trata de forma distinta.

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`.

Ejercicio

2. 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 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.

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 cosas son «la misma cosa», es decir, si son *idénticos*.

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` la misma cosa?

- 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.

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 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 es verdad que podamos sustituir `dep1` por `dep2` (o viceversa) en cualquier parte del programa sin afectar a su funcionamiento.

Es otra forma de decir que **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 si dos objetos son «iguales», y no podemos determinar si son iguales si no podemos observar los efectos de ese cambio.

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)
```

los dos depósitos son distintos.

- 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 la misma cosa 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 distintos o si por el contrario son realmente el mismo depósito. Daría igual.

En general, siempre que no se puedan modificar los objetos, podemos suponer que un objeto compuesto se define 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.

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.

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

Resumiendo:

- Cuando preguntamos si dos objetos son **iguales**, estamos preguntando si tienen 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.

4.2. 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.

En cambio, 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]
```

Sin embargo, 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***.

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*.

4.2.1. El método `__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 primer operando es una instancia de nuestra clase. El segundo operando 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
```

4.2.2. El método `__hash__`

Recordatorio:

- Existen datos *hashables* y datos *no hashables*.
- Los datos *hashables* son aquellos que pueden 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__` de 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__()`.

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

1. Si una clase no define su propio método `__eq__`, tampoco debe definir su propio método `__hash__`.
2. Si una clase define `__hash__`, debe definir también un `__eq__` que vaya a juego con el `__hash__`.
3. Las clases definidas por el programador, de entrada ya traen una implementación predefinida de `__eq__` y `__hash__` (mientras el programador no las cambie por otras) que cumplen que:

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

   x`.__hash__()` devuelve un valor que garantiza que:

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

   Por tanto, si una clase define `__eq__` pero no define `__hash__`, sus instancias no serán *hashables*.
6. Si las instancias de la clase son mutables y ésta define `__eq__`, no debe definir `__hash__`, ya que los objetos mutables no deben ser *hashables*.

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

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.

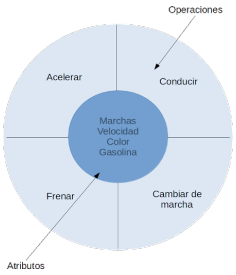
5.1.2. La encapsulación como mecanismo de protección de datos

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.

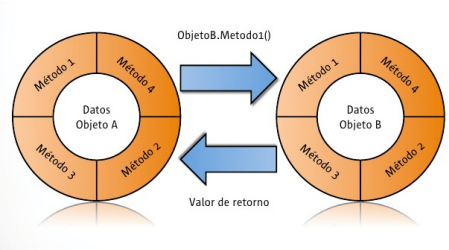
Según esto, podemos imaginar que:

- 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.

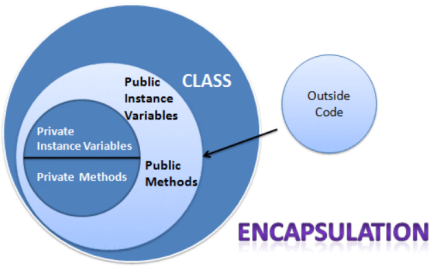
$$\text{Visibilidad} \begin{cases} \text{No se puede acceder desde el exterior, o} \\ \text{Sí se puede acceder desde el exterior} \end{cases}$$

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?

¡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 el atributo `__items` es privado y, por tanto, sólo se puede acceder a él desde el interior de la clase.

El método `get_items` es un *getter* para el atributo `__items`.

En teoría, los únicos métodos con los que podemos modificar el contenido del atributo `__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 al objeto lista contenido 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 al objeto lista, sino una **copia** de la lista:

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

5.1.2.3. Invariantes de clase

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.
- 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 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:

```
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 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).

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.

5.1.2.4. Interfaz y especificación de una clase

Recordemos que 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 los **atributos públicos**.
- La **signatura** de los **métodos públicos**.

Es un concepto puramente sintáctico, porque describe **qué** contiene 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.
- El **invariante** 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.

5.1.2.5. 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
```

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 asertos y sentencias `assert` en puntos adecuados del código fuente de la clase.

5.1.2.6. Un ejemplo completo

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

espec *pila*
parámetros
elemento
operaciones
`pvacia` : $\rightarrow$ *pila*
`apilar` : *pila* $\times$ *elemento* $\rightarrow$ *pila*
parcial `desapilar` : *pila* $\rightarrow$ *pila*
parcial `cima` : *pila* $\rightarrow$ *elemento*
`vacia?` : *pila* $\rightarrow \mathfrak{B}$
var
p : *pila*; x : *elemento*
ecuaciones
`cima`(`apilar`(p, x)) $\doteq x$
`desapilar`(`apilar`(p, x)) $\doteq p$
`vacia?`(`pvacia`) $\doteq V$
`vacia?`(`apilar`(p, x)) $\doteq F$
`cima`(`pvacia`) $\doteq$ *error*
`desapilar`(`pvacia`) $\doteq$ *error*

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

espec *pila*
parámetros
elemento

operaciones
`pila` : → *pila*
`apilar` : *pila* × *elemento* →
parcial `desapilar` : *pila* →
parcial `cima` : *pila* → *elemento*
`vacía?` : *pila* → $\mathfrak{B}$
_ `==` _ : *pila* × *pila* → $\mathfrak{B}$
var
p, p_1, p_2 : *pila*; x : *elemento*
ecuaciones
p_1 `==` $p_2 \doteq$ «p_1 y p_2 tienen los mismos elementos en el mismo orden»
`vacía?`(p) $\doteq$ 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) $\doteq$ «el último elemento apilado en p y aún no desapilado»
`vacía?`(p) $\Rightarrow$ `desapilar`(p) $\doteq$ *error*
`vacía?`(p) $\Rightarrow$ `cima`(p) $\doteq$ *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()` → *pila*

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

`apilar(`*p*: *pila*, *elem*`)` →

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*`)` →

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* $\rightarrow \mathfrak{B}$

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*`)` $\rightarrow \mathfrak{B}$

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

`cima(`*p*: *pila*`)` $\rightarrow$ *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):
        assert not self.vacia()
        self.__elems.pop()

    def cima(self):
        assert not self.vacia()
        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 ope-**

racionales.

6. Miembros de clase

6.1. 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.

Ese valor se guardará en un atributo, pero ese atributo se almacena en la propia clase y está ligado a ella, no a una instancia concreta de dicha clase, ya que es un valor compartido por todos los objetos de la misma clase.

Los atributos que pertenecen y están ligados a la propia clase (en lugar de a instancias concretas) se denominan **atributos de clase**, **variables de clase** o **variables estáticas**.

Por contra, los atributos que hemos visto hasta ahora (los que pertenecen a las instancias) se denominan **atributos de instancia** o **variables de instancia**.

Las variables de clase pertenecen al **espacio de nombres** de la clase.

Los atributos de clase se crean mediante **sentencias de asignación** en el *cuerpo* de la clase, fuera de cualquier definición de método:

```
class Deposito:
    interes = 0.02   # Un atributo 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
```

Este atributo se puede manipular directamente a través de la clase, es decir, pidiéndoselo a la propia clase como si ésta fuese un objeto, usando la sintaxis:

clase`.`*atributo*

```
>>> Deposito.interes
0.02
```

Esto nos indica que los atributos de clase se almacenan en el **diccionario** del **espacio de nombres** de la clase.

Para cambiar el valor de una variable de clase, se le asigna un valor usando la misma sintaxis *clase.atributo*:

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

Las variables de clase 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.

Si necesitamos acceder al valor de una variable de clase dentro de la misma clase, usaremos la misma sintaxis:

```
class Deposito:
    interes = 0.02   # Un atributo 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:
        return self.saldo() * (1 + Deposito.interes)
```

Ejercicio

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

6.2. 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 del espacio de nombres de una clase** y que se ejecuta 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 dentro del espacio de nombres de la clase, para acceder a un método estático hay que usar el operador punto (`.`).

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 en el espacio de nombres 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 (`.`).

Por 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)
```

Bibliografía

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, Mass. : New York: 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.