

Programación orientada a objetos

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Introducción	2
1.1. Recapitulación	2
1.2. La metáfora del objeto	2
1.3. Perspectiva histórica	4
1.4. Lenguajes orientados a objetos	4
2. Conceptos básicos	4
2.1. Clase	4
2.2. Objeto	6
2.2.1. La antisimetría dato-objeto	7
2.3. Identidad	7
2.4. Estado	7
2.5. Atributos	7
2.6. Paso de mensajes	9
2.7. Métodos	9
2.8. Encapsulación	9
2.9. Herencia	10
2.10. Polimorfismo	10
3. Uso básico de objetos	10
3.1. Instanciación	10
3.1.1. <code>new</code>	10
3.1.2. <code>instanceof</code>	10
3.2. Propiedades	10
3.2.1. Acceso y modificación	10
3.3. Referencias	10
3.4. Clonación de objetos	10
3.5. Comparación de objetos	10
3.6. Destrucción de objetos	10
3.6.1. Recolección de basura	10
3.7. Métodos	10
3.8. Constantes	10

4. Clases básicas	10
4.1. Cadenas	10
4.1.1. Inmutables	10
4.1.2. Mutables	10
4.1.3. Conversión a <i>String</i>	11
4.2. Arrays	11
4.3. Clases <i>wrapper</i>	11
4.3.1. Conversiones de empaquetado/desempaquetado (<i>boxing/unboxing</i>)	11
5. Lenguaje UML	11
5.1. Diagramas de clases	11
5.2. Diagramas de objetos	11
5.3. Diagramas de secuencia	11

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 las demás partes del programa les envían mensajes.

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

Definición:

La **programación orientada a objetos** es un paradigma de programación en el que los programas son vistos como formados por entidades llamadas **objetos** que recuerdan su propio **estado 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 el estado local y el comportamiento de una manera que abstrae la complejidad de ambos.
- Los objetos se comunican entre sí y se obtienen resultados útiles como consecuencia de su interacción.
- Los objetos no sólo transmiten mensajes, sino que también comparten el comportamiento entre otros objetos del mismo tipo y heredan características de otros tipos relacionados.

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

1.3. Perspectiva histórica

1.4. Lenguajes orientados a objetos

2. Conceptos básicos

2.1. Clase

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 Depósito:
    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
```

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

```
class <nombre>:
    <sentencia>+
```

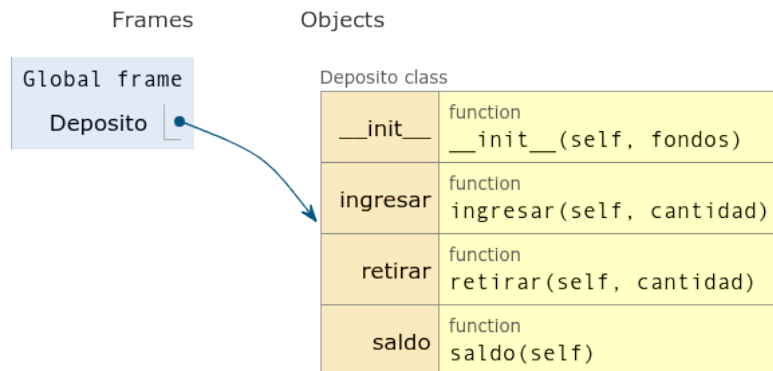
Todas las definiciones que se hagan dentro de la clase serán **locales** a ella, al encontrarse dentro del ámbito de dicha clase.

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

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

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

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

La clase `Deposito` en memoria

2.2. Objeto

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.
- El tipo `int` es un tipo primitivo del lenguaje, por lo que el número 5 no es ningún objeto, sino un dato primitivo.

En Python podemos instanciar una clase (creando así un nuevo objeto) llamando a la clase como si fuera una función, del mismo modo que 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__`.

2.2.1. 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.

2.3. Identidad

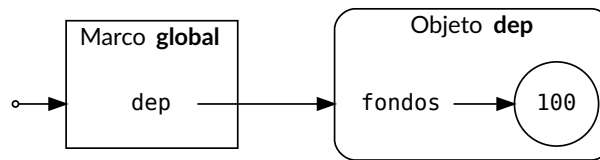
2.4. Estado

2.5. Atributos

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

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

Cada vez que se crea un objeto, se le asocia una zona de memoria que almacena los atributos del mismo.

Objeto `dep` y su atributo `fondos`

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

```

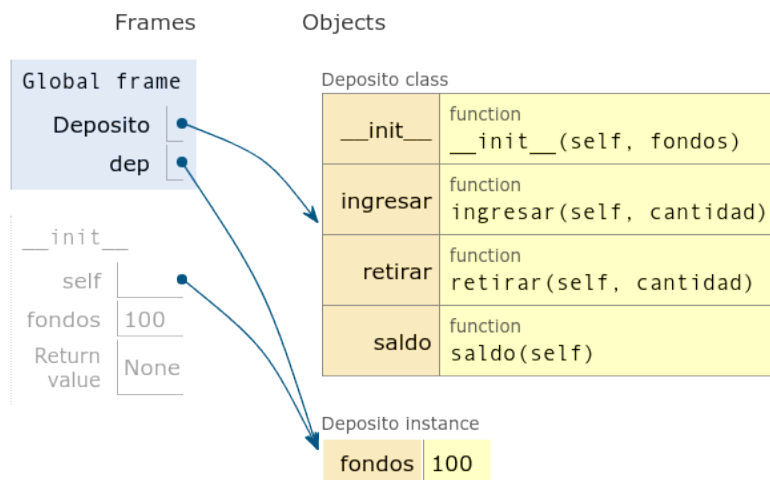
class Deposito:
    def __init__(self, fondos):
        self.fondos = fondos

    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

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.

No obstante, en la práctica puede resultar útil y, de hecho, dentro de la definición de una clase es necesario hacerlo así para poder acceder al valor de un atributo de la misma.

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

objeto.atributo

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

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

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

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

Otro ejemplo: si tenemos el número complejo `4 + 3j`, podemos preguntar cuál es su parte real y su parte imaginaria (que son atributos del objeto):

```
>>> c = 4 + 3j
>>> c
(4+3j)
>>> c.real
4.0
>>> c.imag
3.0
```

Pero esos atributos no se pueden modificar directamente, ya que son de *sólo lectura*:

```
>>> c.real = 9.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: readonly attribute
```

2.6. Paso de mensajes

El paso de mensajes se realiza ahora invocando

2.7. Métodos

2.8. Encapsulación

2.9. Herencia

2.10. Polimorfismo

3. Uso básico de objetos

3.1. Instanciación

3.1.1. `new`

3.1.2. `instanceof`

3.2. Propiedades

3.2.1. Acceso y modificación

3.3. Referencias

3.4. Clonación de objetos

3.5. Comparación de objetos

3.6. Destrucción de objetos

3.6.1. Recolección de basura

3.7. Métodos

3.8. Constantes

4. Clases básicas

4.1. Cadenas

4.1.1. Inmutables

4.1.1.1. `String`

4.1.2. Mutables

4.1.2.1. `StringBuffer`

4.1.2.2. **StringBuilder**

4.1.2.3. **StringTokenizer**

4.1.3. **Conversión a String**

4.2. **Arrays**

4.3. **Clases *wrapper***

4.3.1. Conversiones de empaquetado/desempaquetado (*boxing/unboxing*)

5. **Lenguaje UML**

5.1. **Diagramas de clases**

5.2. **Diagramas de objetos**

5.3. **Diagramas de secuencia**