Introducción a la programación orientada a objetos

Programación para el Cálculo Científico

Máster Universitario en Cálculo y Modelización Científica
Universidad de Alicante
Curso 2024-2025

Índice

Introducción

Definición

- La *programación orientada a objetos* (POO) es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas informáticos
- La aplicación entera se reduce a un conjunto de objetos y sus relaciones
- Python es un lenguaje orientado a objetos, aunque también permite programación imperativa (*procedimental*)
- Cambia el enfoque a la hora de diseñar los programas...
- ... ¡pero todo lo que has aprendido hasta ahora te sigue valiendo!

Clases y objetos (1/2)

- En Python ya hemos usado clases y objetos:

```
i = 4          # definimos una variable de tipo entero
s = "Hola"     # definimos una variable (objeto)
               # de tipo (clase) string
```

- Una *clase* (o tipo compuesto) es un modelo para crear objetos de esa clase
- Un *objeto* de una determinada clase se denomina una *instancia* de la clase
- En el ejemplo anterior, `s` es una instancia/objeto de la clase `string`
- Las clases son similares a los tipos simples, aunque permiten muchas más funcionalidades

Clases y objetos (2/2)

- Una clase contiene datos y una serie de funciones que manipulan esos datos, llamadas *funciones miembro* o *métodos*
- Se puede controlar qué datos/métodos son visibles (o públicos) y cuáles están ocultos (privados)*
- Las funciones miembro pueden acceder a los datos públicos y privados de su clase
- Las funciones miembro públicas definen el *interface* de la clase

*Aunque en Python la diferencia es poco estricta.

Conceptos básicos

Conceptos básicos

- Principios en los que se basa el diseño orientado a objetos:
  - Abstracción
  - Encapsulación
  - Modularidad
  - Herencia
  - Polimorfismo

Abstracción

- La *abstracción* denota las características esenciales de un objeto y su comportamiento
- Cada objeto puede realizar tareas, informar y cambiar su estado, comunicándose con otros objetos en el sistema sin revelar cómo se implementan estas características
- El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevas clases
- El proceso de abstracción tiene lugar en la fase de diseño

Encapsulación

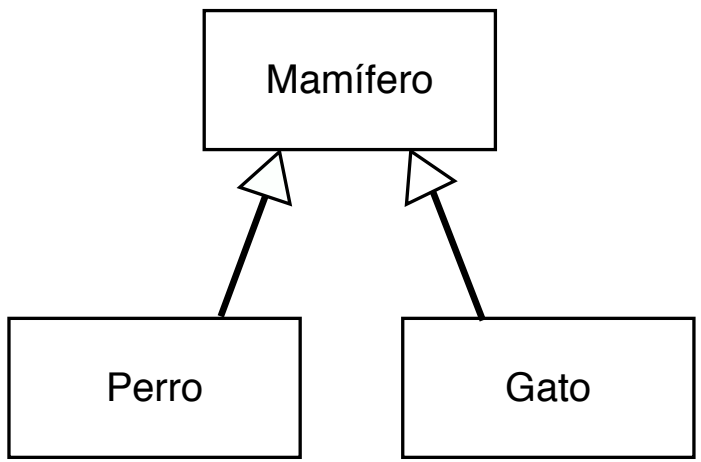
- La *encapsulación* significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad al mismo nivel de abstracción
- La *interfaz* es la parte del objeto que es visible (pública) para el resto de los objetos: conjunto de métodos y datos de los cuales disponemos para comunicarnos con un objeto
- Cada objeto oculta su implementación (cómo lo hace) y expone una interfaz (qué hace)
- La encapsulación protege a las propiedades de un objeto contra su modificación: solamente los propios métodos del objeto pueden acceder a su estado

Modularidad

- Se denomina *modularidad* a la propiedad que permite subdividir una aplicación en partes más pequeñas (*módulos*) tan independientes como sea posible
- Estos módulos se pueden utilizar por separado, pero tienen conexiones con otros módulos
- Generalmente, cada clase se implementa en un módulo independiente, aunque clases con funcionalidades similares también pueden compartir módulo

Herencia (1/2)

- Las clases se pueden relacionar entre sí formando una jerarquía de clasificación
- La *herencia* permite definir una nueva clase a partir de otra
- Se aplica cuando hay suficientes similitudes y la mayoría de las características de la clase existente son adecuadas para la nueva clase
- En este ejemplo, las *subclases* `Perro` y `Gato` heredan los métodos y atributos especificados por la *superclase* `Mamífero`:



Herencia (2/2)

- La herencia nos permite adoptar características ya implementadas por otras clases
- Facilita la organización de la información en diferentes niveles de abstracción
- Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen
- Los objetos derivados pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo
- Cuando un objeto hereda de más de una clase se dice que hay *herencia múltiple*

Polimorfismo

- El *polimorfismo* es la propiedad según la cual una misma expresión hace referencia a distintas acciones
- Por ejemplo, un método `desplazar` puede referirse a acciones distintas si se trata de un avión o de un coche
- Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre

POO en Python

Declaración e implementación (1/2)

- Los objetos se definen usando clases y las variables que se definen en ella son propiedades comunes de ese objeto. Por ejemplo, consideremos una nueva clase llamada Star:

```
class Star:
'''Clase para estrellas'''
    def __init__(self, name):
        self.name = name
    def __str__(self):  # Método especial que se llama
                        # cuando se hace print
        return "Estrella {}".format(self.name)
```

- `__init__` es el *constructor* de la clase. Contiene el código necesario para inicializar el objeto. Le hemos puesto un parámetro `name` pero puede tener los que queramos (o ninguno)
- `self`: variable especial que hace referencia al objeto que estamos creando o manipulando. Por convenio se utiliza `self`, pero se puede poner cualquier otro nombre

Declaración e implementación (2/2)

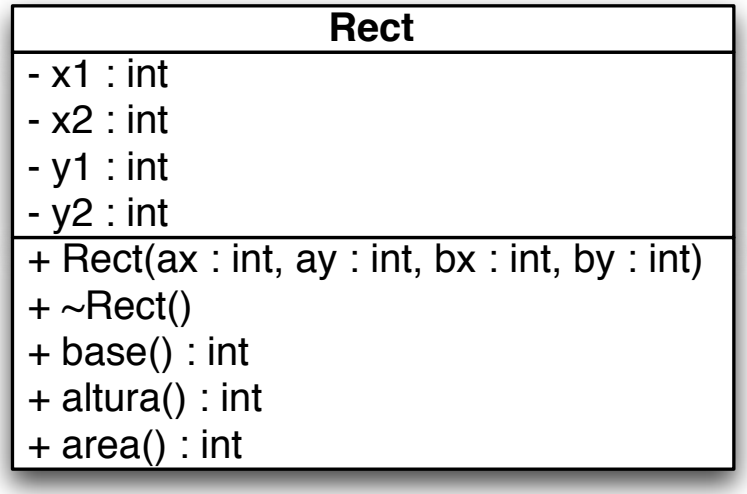
Si el código anterior lo guardamos en un archivo de nombre `star.py` podemos crear objetos de tipo `Star` en cualquier otro programa

```
import star  # importa el código del archivo star.py
estrella = star.Star("Altair")    # estrella es una instancia
                                  # de Star con nombre Altair
print(estrella)    # imprime lo que devuelve estrella.__str__()
```

Antes de añadir nuevas funcionalidades a las clases vamos a ver una manera de gráfica de poner orden a las relaciones entre objetos

Diagrama UML (1/3)

- Un *diagrama UML* permiten describir las clases y relaciones entre clases en un diseño orientado a objetos:



- El – delante de un atributo o método indica que es privado
- El + indica que es un atributo o método público
- La línea horizontal separa los atributos (parte superior) de los métodos (parte inferior)

Diagrama UML (2/3)

- Traducción a código Python del diagrama UML anterior:

```
# rect.py (declaración de la clase)
class Rect:
    def __init__(self, ax, ay, bx, by):
        self._x1 = ax
        self._y1 = ay
        self._x2 = bx
        self._y2 = by
    def base(self):
        return self._x2 - self._x1
    def altura(self):
        return self._y2 - self._y1
    def area(self):
        return self.base()*self.altura()
```

Diagrama UML (3/3)

- `~Rect()` no se utiliza en Python porque su tarea la realiza automáticamente el *recolector de basura* (es el llamado *destructor* de la clase)
- En Python no hay diferencia entre atributos/métodos públicos o privados. Todo es público
- Se utiliza el convenio de usar el guión bajo (_) para indicar que un atributo u objeto debería tratarse como privado (aunque no lo es)

Accesores

- No es conveniente acceder directamente a los datos miembro de una clase (principio de encapsulación)
- Lo normal es definirlos como privados y acceder a ellos implementando métodos *set/get/is* (llamados *accesores*):

- Los accesores `set` nos permiten controlar que los valores de los atributos sean correctos

Constructor (1/7)

- El *constructor* se invoca automáticamente cuando se crea un objeto de la clase
- Las clases deben tener al menos un método constructor
- En el constructor se crean los atributos del objeto anteponiendo a su nombre el prefijo `self.`
- Una clase puede tener varios constructores con parámetros distintos (el constructor puede *sobrecargarse*)
- La sobrecarga es un tipo de polimorfismo

Constructor (2/7)

- Ejemplos de constructor:

```
class Fecha:
    def __init__(self): # Sin parámetros
        self.dia = 1
        self.mes = 1
        self.anyo = 1900

    def __init__(self, d, m, a): # Con tres parámetros
        self.dia = d
        self.mes = m
        self.anyo = a
```

- Llamadas al constructor:

```
f = Fecha()
f = Fecha(10,2,2010)
```

Constructor (3/7)

- Los constructores (al igual que otras funciones) pueden tener parámetros por defecto:

```
class Fecha:
  ...
  def __init__(self, d=1, m=1, a=1900):
      self.dia = d
      self.mes = m
      self.anyo = a
  ...
```

- Con este constructor podríamos crear objetos de varias formas:

```
f = Fecha()  # dia = 1, mes = 1, anyo = 1900
f = Fecha(10,2,2010)   # dia = 10, mes = 2, anyo = 2010
f = Fecha(10)   # dia = 10, mes = 1, anyo = 1900
f = Fecha(18,5)   # dia = 18,  mes = 5, anyo = 1900
```

Constructor (4/7)

- Los parámetros por defecto del ejemplo anterior se mostrarían de la siguiente manera en un diagrama UML:

Fecha
- dia: int - mes: int - anyo: int
+ Fecha (dia: int=1, mes: int=1, anyo: int=1900) ...

Constructor (5/7)

- Si los parámetros que se le pasan al constructor son incorrectos no debería de crearse el objeto
- Esto se puede controlar mediante el uso de *excepciones*:
  - Podemos lanzar una excepción con `raise` para indicar que se ha producido un error
  - Podemos capturar una excepción con `try/except` para reaccionar ante el error
- Si se produce una excepción y no la capturamos, el programa terminará inmediatamente
- Las excepciones sólo deben usarse cuando no hay otra opción (por ejemplo, en los constructores)

Constructor (6/7)

- Ejemplo de uso de excepciones:

```
def root(n):
    if type(n) not in (int, float) or n < 0:
        raise ValueError # Lanza la excepción y termina
    return n**0.5;

try: # Intentamos ejecutar estas instrucciones
    result = root(-1)    # Provoca una excepción
    print(result)        # Esta línea no se ejecuta
except ValueError: # Si hay una excepción se captura aquí
    print("No se puede calcular la raiz cuadrada")
```

Constructor (7/7)

- Ejemplo de constructor con excepción:

```
class Coordenada:
    def __init__(self, cx, cy):
        if cx >= 0 and cy >= 0:
            self.x = cx
            self.y = cy
        else:
            raise ValueError  # o la excepción que
                              # consideremos adecuada
try:
    c = Coordenada(-2,4) # Este objeto no llega
                         # a crearse
except ValueError:
    print("Coordenada incorrecta")
```

Operador de asignación

- El *operador de asignación* (=) permite una asignación directa de dos objetos:

```
f1 = Fecha(10, 2, 2011) # Constructor
f2 = f1
# f2 es un alias de f1, ambas variables representan
# el mismo objeto
```

- Si queremos crear una copia atributo a atributo

```
import copy

f1 = Fecha(10, 2, 2011) # Constructor
f2 = copy.copy(f1)
# f2 es una copia atributo a atributo de f1
# pero son dos objetos distintos
```

Atributos y métodos de clase (1/3)

- Los *atributos de clase* tienen el mismo valor para todos los objetos de la clase (son como variables globales para la clase)
- También se llaman atributos *estáticos*
- Se definen en el ámbito principal de la clase

```
class Star:
    # Numero total de estrellas
    num_stars = 0

    def __init__(self, name):
        self.name = name
        Star.num_stars += 1
        # Cada vez que creamos una estrella actualizamos
        # el número total
```

Atributos y métodos de clase (2/3)

- Los *métodos de clase* son métodos que se usan para realizar operaciones en el ámbito de clase y no al generar una instancia
- Sólo pueden acceder a los atributos y métodos de clase
- Para crear un método de clase, debemos usar el decorador `@classmethod` y en el método debemos añadir como primer parámetro `cls`

```
class Perro:
  pesoPromedio = 30

  def __init__(self, peso):
      self.peso = peso

  def getPeso():
      return self.peso

  @classmethod
  def getPesoPromedio(cls):
      return cls.pesoPromedio
```

Atributos y métodos de clase (3/3)

- Se representan subrayados en los diagramas UML
- En este ejemplo hay dos atributos estáticos o de clase (`IVA` y `nextId`) y un método de clase (`getNextId`):

Factura
-nextId: int = 1 +IVA: const int = 21 -fecha: string -id: int
+Factura(c:Clente. fecha:string) +anyadirLinea(cant:int, desc:string, prec:float): void -getNextId(): int +__str__(): string

Métodos estáticos

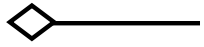
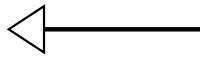
- Los *métodos estáticos* son un tipo de método de clase que no necesita acceder a los atributos de clase pero que interesa tener agrupados dentro de la clase
- En este caso no necesitaremos ningún parámetro principal como sucede con los métodos normales o los de clase
- Para declarar un método estático, solo necesitamos usar el decorador `@staticmethod` y nuestro método estático.

```
class Math:
    @staticmethod
    def sumar(num1, num2):
        return num1 + num2

    @staticmethod
    def restar(num1, num2):
        return num1 - num2

print(Math.sumar(5, 7))
print(Math.restar(9, 3))
```

Relaciones

Relaciones entre objetos

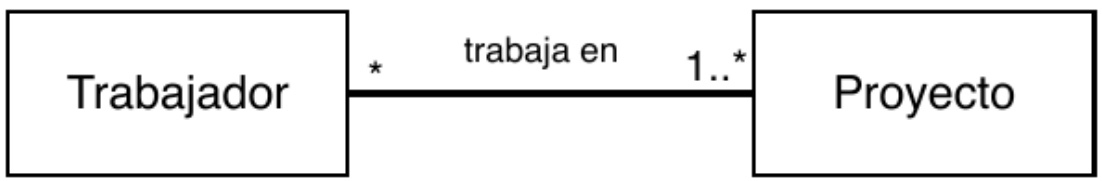
- Principales tipos de relaciones entre objetos y clases:

Entre objetos	Asociación	
	Agregación	
	Composición	
	Uso	
Entre clases	Generalización	

- La mayoría de las relaciones posee cardinalidad:
  - Uno o más: 1..$*$ (1..*n*)
  - Cero o más: $*$
  - Número fijo: *m*

Asociación

- La asociación expresa una relación (unidireccional o bidireccional) general entre los objetos instanciados a partir de las clases conectadas
- la relación de la figura siguiente expresa que un trabajador está vinculado a uno o más proyectos y que en un proyecto pueden trabajar cero o más trabajadores

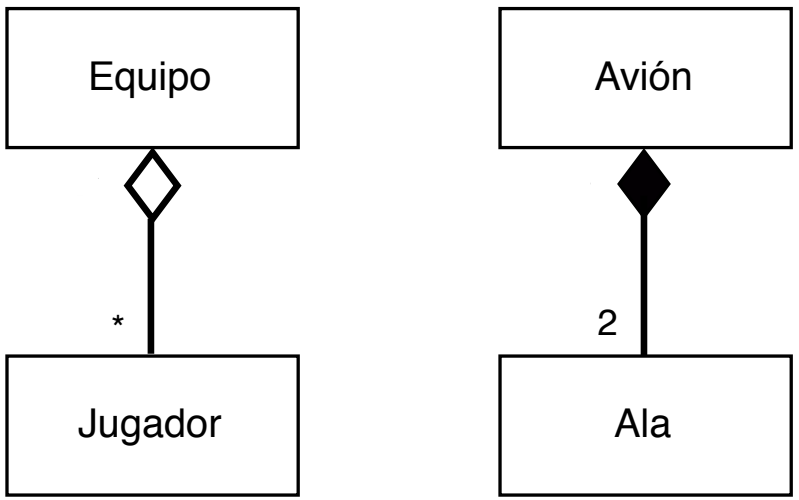


Agregación y composición (1/6)

- *Agregación* y *composición* son relaciones todo-parte en las que un objeto forma parte de la naturaleza de otro
- Son relaciones asimétricas
- La diferencia entre agregación y composición es la fuerza de la relación: la agregación es una relación más débil que la composición

Agregación y composición (2/6)

- En la composición, cuando se destruye el objeto contenedor también se destruyen los objetos que contiene
  - Ej: el ala forma parte del avión y no tiene sentido fuera del mismo (si vendemos un avión, lo hacemos incluyendo sus alas)
- En el caso de la agregación, no ocurre así
  - Ej: podemos vender un equipo, pero los jugadores pueden irse a otro club (no desaparecen con el equipo)

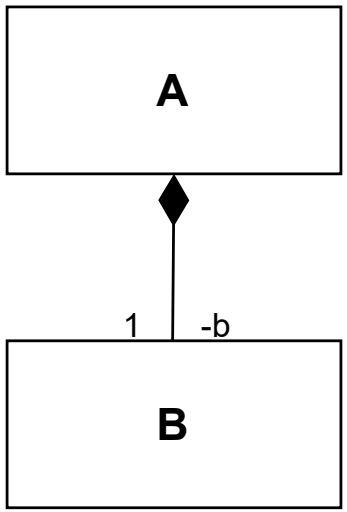


Agregación y composición (3/6)

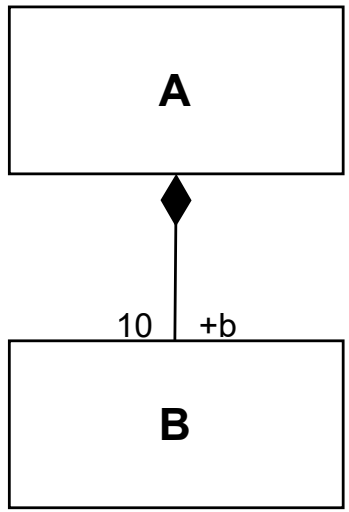
- Algunas relaciones pueden ser consideradas como agregaciones o composiciones en función del contexto en que se utilicen
  - Ej: la relación entre bicicleta y rueda
- Algunos autores consideran que la única diferencia entre ambos conceptos radica en su implementación: una composición sería una “agregación por valor”

Agregación y composición (4/6)

- Implementación de la composición:



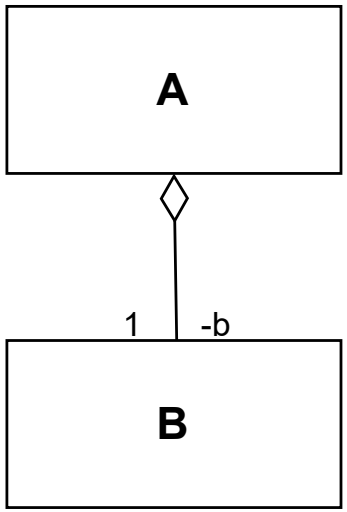

```
class A:
    def __init__(self):
        self._b = B()
        ...
```



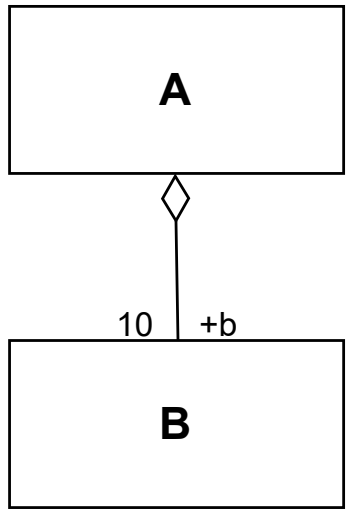

```
class A:
    def __init__(self):
        self.b = []
        for i in range(10):
            x = B()
            self.b.append(x)
        ...
```

Agregación y composición (5/6)

- Implementación de la agregración:




```
class A:
    def __init__(self, c):
        # c = B() antes de crear A
        self._b = c
    ...
```




```
class A:
    def __init__(self, c):
        # c es una lista de 10 objetos B
        self.b = []
        for x in c:
            self.b.append(x)
    ...
```

Agregación y composición (6/6)

- Ejemplo de implementación de la agregración:

```
class A:
    def __init__(self, b):
        self.b = b
        ...

class B:
    def __init__(self):
        ...

b = B()
a = A(b)
...
```

Uso (1/2)

El uso es una relación no persistente (tras la misma, se termina todo contacto entre los objetos). Diremos que una clase A usa una clase B cuando:

- Invoca algún método de la clase B
- Tiene alguna instancia de la clase B como parámetro de alguno de sus métodos
- Accede a sus variables privadas (esto sólo se puede hacer si son clases amigas)

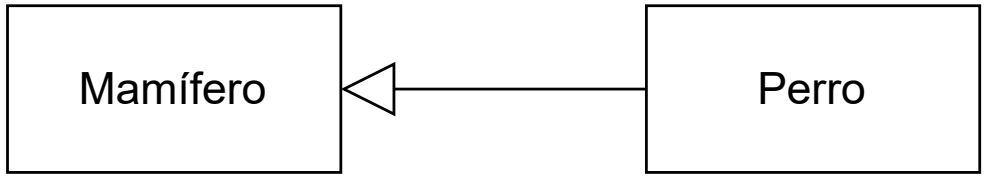
Uso (2/2)

- Ejemplo de uso:


```
class Coche:
    ...
    def repostar(self, g, litros):
        importe = g.dispensarGaso(litros,self.tipo)
        self.lgaso += litros
        return import
    ...
class Gasolinera:
    ...
    def dispensarGaso(self, litros, tipoC):
        ...
        return importe
    ...
```

Generalización / Herencia en Python (1/8)

- Implementar una clase derivada (o subclase)




```
class Perro(Mamifero):
    ...
```

- Todas las clases en Python heredan de la clase base `object`, primera en la jerarquía de clases
- Python admite herencia múltiple
- La clase Perro contiene todos los atributos y métodos de la clase Mamífero.
- Además, estos métodos se pueden *sobreescribir* (pueden hacer las cosas de diferente forma en la clase derivada)

Generalización / Herencia en Python (2/8)

- Ejemplo clase padre (superclase)

```
class Mamifero(object):  # parámetro oject no necesario
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad
    # Método genérico pero con implementación particular
    def moverse(self):
        # Método vacío
        pass
    # Método genérico con la misma implementación
    def describeme(self):
        print("Soy un Mamifero del tipo",
              type(self).__name__)
```

- Todos los Mamíferos son de alguna especia y tienen una edad
- Cada animal se moverá de una forma (volar, nadar, caminar)
- Todos se describirán de la misma forma

Generalización / Herencia en Python (3/8)

- Ejemplo clase derivada (subclase)

```
# Perro hereda de Mamifero
class Perro(Mamifero):
    pass

mi_perro = Perro('canido', 10)
mi_perro.describeme()
# Soy un Mamifero de tipo Perro
```

- Perro ya dispone de todo lo que tiene Mamífero

Generalización / Herencia en Python (4/8)

- Pero podemos añadir y modificar cosas nuevas...

```
class Perro(Mamífero):
    def moverse(self):
        print("Caminando con 4 patas")

class Delfin(Mamífero):
    def moverse(self):
        print("Nadando por el mar")

class Vampiro(Mamífero):
    def moverse(self):
        print("Volando")

    # Nuevo método
    def chupar(self):
        print("¡Chupando sangre!")
```

Generalización / Herencia en Python (5/8)

- Cada clase derivada cambia el comportamiento de los métodos sobreescritos

```
mi_perro = Perro('canido', 10)
mi_delfin = Vaca('delfinido', 23)
mi_vampiro = Vampiro('desmodus rotundus', 1)

mi_perro.moverse()
# Caminando con 4 patas
mi_delfin.moverse()
# Nadando por el mar
mi_delfin.describeme()
# Soy un Mamifero del tipo Delfin
mi_vampiro.describeme()
# Soy un Mamifero del tipo Vampiro
mi_vampiro.chupar()
# ¡Chupando sangre!
```

Generalización / Herencia en Python (6/8)

- Accediendo a los métodos de la clase padre: `super()`

```
class Perro(Mamifero):
    def __init__(self, especie, edad, dueño):
        # Alternativa 1
        # self.especie = especie
        # self.edad = edad
        # self.dueño = dueño

        # Alternativa 2
        super().__init__(especie, edad)
        self.dueño = dueño

mi_perro = Perro('canido', 7, 'Luis')
print(mi_perro.especie)     # canido
print(mi_perro.edad)        # 7
print(mi_perro.dueño)       # Luis
```

Generalización / Herencia en Python (7/8)

- Herencia múltiple y método `__mro__`

```
class Clase1:
    pass
class Clase2:
    pass
class Clase3(Clase1, Clase2):
    pass

print(Clase3.__mro__)
# (<class '__main__.Clase3'>, <class '__main__.Clase1'>, <class
    '__main__.Clase2'>, <class 'object'>)
```

Generalización / Herencia en Python (8/8)

Métodos especiales (hay más...)

https://docs.python.org/es/3/reference/datamodel.html#special-method-names

Método	Operador
`__add__`	+
`__sub__`	-
`__mul__`	*
`__div__`	/
`__floordiv__`	//
`__lt__`	<
`__le__`	<=
`__eq__`	==
`__ne__`	!=
`__gt__`	>
`__ge__`	>=

El API de Matplotlib

El API de Matplotlib

- Exploremos
  `https://matplotlib.org/stable/api/index.html`
- Tutorial interesante:
  `https://www.studytonight.com/matplotlib/matplotlib-object-oriented-interface`

Un ejemplo

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.figure import Figure

class WatermarkFigure(Figure): # A figure with a text watermark
    def __init__(self, *args, watermark=None, **kwargs):
        super().__init__(*args, **kwargs)
        if watermark is not None:
            bbox = dict(boxstyle='square', lw=3, ec='gray',
                        fc=(0.9, 0.9, .9, .5), alpha=0.5)
            self.text(0.5, 0.5, watermark,
                      ha='center', va='center', rotation=30,
                      fontsize=40, color='gray', alpha=0.5,
                      bbox=bbox)
x = np.linspace(-3, 3, 201)
y = np.tanh(x) + 0.1 * np.cos(5 * x)
plt.figure(FigureClass=WatermarkFigure, watermark='draft')
plt.plot(x, y)
plt.show()
```

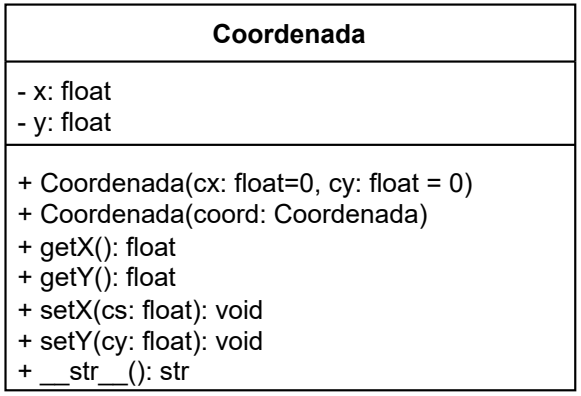
Ejercicios

Consideraciones generales

- Cada clase que aparezca en los ejercicios se debe escribir en un fichero distinto a no ser que el enunciado indique otra cosa.
- En caso de que se requiera un programa principal éste tendrá el nombre `principal_i.py`, donde `i` es el número del ejercicio.
- Todos los ejercicios se guardarán en una carpeta de nombre `Ejercicios` en tu repositorio de GitHub.

Ejercicio 1

Implementa la clase del siguiente diagrama:



Debes crear el fichero `Coordenada.py` junto con un programa `principal_1.py`, en el que se debe pedir al usuario dos números y crear con ellos una coordenada para imprimirla con la función `print` en el formato `(x,y)`. Escribe el código necesario para que cada método sea utilizado al menos una vez.

Ejercicio 2

Vamos a crear una clase llamada `Persona`. Sus atributos son: `nombre`, `edad` y `DNI`. Construye los siguientes métodos para la clase:

- Un constructor, donde los datos pueden estar vacíos (en cuyo caso se asumirá que `nombre = "", edad = 0, DNI = ""`).
- Los setters y getters para cada uno de los atributos. Hay que validar las entradas de datos.
- Un método `__str__()` que convierta en cadena todos los atributos.
- `esMayorDeEdad()`: Devuelve un valor lógico indicando si es mayor de edad.

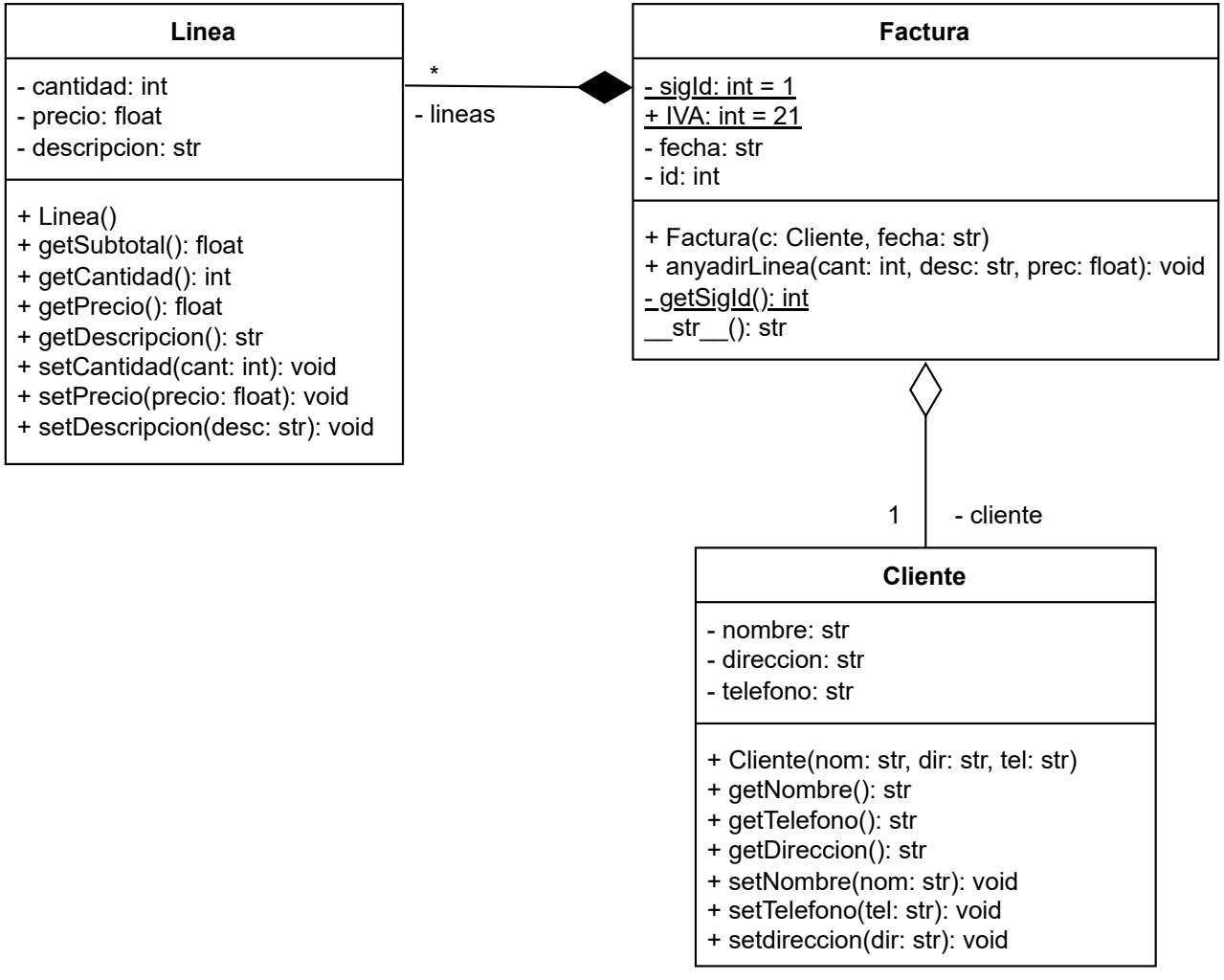
Ejercicio 3

Crea una clase llamada `Cuenta` que tendrá los siguientes atributos: `titular` (que es una persona) y `cantidad` (puede tener decimales). El `titular` será obligatorio y la `cantidad` es opcional. Construye los siguientes métodos para la clase:

- Un constructor, donde la cantidad puede estar vacía.
- Los setters y getters para cada uno de los atributos. El atributo no se puede modificar directamente, sólo ingresando o retirando dinero.
- Un método `__str__()`
- `ingresar(cantidad)`: se ingresa una cantidad a la cuenta, si la cantidad introducida es negativa, no se hará nada.
- `retirar(cantidad)`: se retira una cantidad a la cuenta. La cuenta puede estar en números rojos.

Ejercicio 4

Implementa el código correspondiente al siguiente diagrama UML:



Ejercicio 5

Se debe hacer un programa (`principal_5.py`) que cree una nueva factura, añada un producto y lo imprima. Desde el constructor de `Factura` se llamará al método `getSigId`, que devolverá el valor de `sigId` y lo incrementará. Ejemplo de salida al imprimir una factura:

```
Factura nº: 12345
Fecha: 18/4/2011

Datos del cliente
-----------------
Nombre: Agapito Piedralisa
Dirección: c/ Río Seco, 2
Teléfono: 123456789

Detalle de la factura
---------------------
Línea;Producto;Cantidad;Precio ud.;Precio total
--
1;Ratón USB;1;8.43;8.43
2;Memoria RAM 2GB;2;21.15;42.3
3;Altavoces;1;12.66;12.66

Subtotal: 63.39 €
IVA (21%): 13.3119 €
TOTAL: 76.7019 €
```

Ejercicio 6

Vamos a crear una clase `Conjunto` que se comporte como ese concepto matemático (una secuencia de elementos que no se repiten y cuyo orden carece de importancia). Aunque Pyhton ya dispone del tipo de datos `set`, para este ejercicio está prohibido usarlo. Debes implementar los siguientes métodos:

- Un constructor que construya un conjunto a partir de los elementos de una secuencia
- operador '|': debe devolver el conjunto unión de los operandos de tipo `Conjunto`.
- operador '`&`': análogo al anterior pero con la intersección.
- operador '−': devuelve la diferencia entre el conjunto de la izquierda y el de la derecha.
- operador '`+`': idem con la diferencia simétrica.
- `cardinal`: devuelve el número de elementos en el conjunto
- `__str__`: para escribir el conjunto con la notación habitual

Ejercicio 7

Papel, bolígrafo, marcador

- Escribir una clase `Papel` que contenga un `texto`, un método `escribir`, que reciba una cadena para agregar al `texto`, y el método `__str__` que imprima el contenido del `texto`.
- Escribir una clase `Boligrafo` que contenga una cantidad de `tinta`, y un método `escribir`, que reciba un `texto` y un `Papel` sobre el cual escribir. Cada letra escrita debe reducir la cantidad de `tinta` contenida. Cuando la tinta se acabe, debe lanzar una excepción.
- Escribir una clase `Marcador` que herede de `Boligrafo`, y agregue el método `recargar`, que reciba la cantidad de `tinta` a agregar.

Ejercicio 8

Juego de rol

- Escribir una clase `Personaje` que contenga los atributos `vida`, `posicion` y `velocidad`, y los métodos `recibir_ataque`, que reduzca la vida según una cantidad recibida y lance una excepción si la `vida` pasa a ser menor o igual que cero, y `mover` que reciba una dirección y se mueva en esa dirección la cantidad indicada por `velocidad`.
- Escribir una clase `Soldado` que herede de `Personaje`, y agregue el atributo `ataque` y el método `atacar`, que reciba otro personaje como parámetro, al que le debe hacer el daño indicado por el atributo `ataque`.
- Escribir una clase `Campesino` que herede de `Personaje`, y agregue el atributo `cosecha` y el método `cosechar`, que devuelva la cantidad cosechada.

Ejercicio 9

Cambia la marca de agua del ejemplo del API de Matplotlib por una lo más parecida posible a la de la figura. Para ello debes construir una nueva clase de herede de `matplotlib.figure.Figure` y que introducirás en la posición adecuada al crear el objeto `plt.figure`. Todo el código de este ejercicio debe estar en el archivo `ejercicio_9.py`.