Tema 7. Introducción a la POO

December 10, 2020

1 Introducción a la POO

- La Programación Orientada a Objetos (POO) es una forma de organizar el código de un programa complejo
- Sin POO un programa complejo se organiza en distintos ficheros (módulos) que contienen la definición de una serie de funciones y un programa principal en el que importo los otros módulos y voy invocando a las funciones. Esto se conoce como *programación estructurada* o *programación modular*
- La programación estructurada agrupa las funciones por su funcionalidad
    - En el ejemplo del proyecto final de este año, las funciones que mueven el lemming van en un módulo, las que dibujan van en otro, las que se encargan de procesar la interacción con el usuario van en otro, etc.
- En POO divido el programa creando un módulo (clase) para cada entidad relevante en mi problema.
    - En el proyecto final: creo una clase con todo lo relativo a lemmings, otra con todo lo relacionado con plataformas, otra con herramientas
- Cada clase contiene la información relevante para ese objeto (variables) y todo lo que ese objeto puede hacer (funciones)
    - A las variables que declaro dentro en una clase se les llama atributos o campos
    - A las funciones que declaro dentro de una clase se les llama métodos
- Analizo el problema teniendo en cuenta:
    - Las entidades más relevantes que aparecen en ese problema (clases)
    - La información que necesito para caracterizarlas (atributos)
    - Lo que pueden hacer esas entidades (métodos)
- En el proyecto final:
    - Entidades: plataformas, lemmings, herramientas, etc
    - Información de un lemming: x, y, dirección, imagen, etc
    - Métodos de un lemming: mover, subir, bajar, etc

2 Clase

- Una clase es una definición genérica de la información (atributos) y las funciones (métodos) que van a tener una serie de objetos
- En una primera aproximación algo burda se pueden ver como una especie de diccionarios genéricos
    - Recordad los 3 diccionarios (maria, pedro, miguel) que había que hacer en un ejercicio de una semana anterior. Aunque los 3 tenían las mismas claves había que repertir su definición 3 veces.
- Para definir una clase uso `class NombreDeClase`
- Regla de estilo: camelloAlto
- A continuación pongo los atributos (aunque veremos luego que hay una forma mejor de hacerlo). Los atributos son todas las variables que declaro dentro de la clase (como si fueran las claves de un diccionario)
- Es como si estuviera declarando un nuevo tipo: la clase es el nuevo tipo y los objetos las variables de ese tipo
- Generalmente se usa un módulo para declarar la clase (cada clase tiene su fichero propio) y luego tengo un módulo en el que declaro variables de esa clase
- Para declarar una variable de esa clase pongo: `nombre_variable = NombreClase()`
- Una variable de una clase se denomina *objeto*
- Para darle valor a un atributo de un objeto uso `nombre_variable.atributo = valor`

[1]:
```
class Estudiante:
    """ Esta clase representa un estudiante con sus notas"""
    # Esto son los atributos de la clase
    nombre: str
    nota: list
    ejerciciosSemanales: list
```

[2]:
```
# Creo una variable de la clase estudiante
# A la variable estu se le llama objeto
estu = Estudiante()
# Le doy valor a sus atributos
estu.nombre = "Pepe"
estu.nota = [10, 4, 6]
estu.ejerciciosSemanales = [4, 7, 8, 9, 3]
# Imprimo el valor de uno de sus atributos
print(estu.nota)
# Declaro otro objeto de esa clase
estu2 = Estudiante()
# Le doy valor a un atributo
estu2.nombre = "Maria del Mar"
```

```
[10, 4, 6]
```

3 Métodos

- Un método es una función que se declara dentro de una clase
- La única diferencia con una función normal es que está obligado a tener al menos un parámetro
- Este parámetro es un parámetro especial y aunque podría tener cualquier nombre se le suele llamar `self` y es siempre el primero del método
- Mediante este parámetro puedo acceder a los atributos del objeto dentro del método
- Para invocar ese método necesito haber creado un objeto
- Cuando se llama al método hay que dar valor a todos los parámetros menos a `self`
- Para llamar a un método pongo `nombre_objeto.nombre_método(parametros_sin_contar_self)`

[3]:
```python
class Estudiante:
    """ Esta clase representa un estudiante con sus notas. Incluye
    un método para calcular la media de los ejercicios semanales"""
    # Esto son los atributos de la clase
    nombre: str
    nota: list
    ejerciciosSemanales: list

    def media(self):
        """ Método que calcula la media de los ejercicios semanales.
        Como todo método debe tener al menos 'self' como parámetro"""
        # Esto es una variable local del método
        med = 0
        # Para acceder en un método a un atributo del objeto uso
        # self.nombre_atributo
        for nota in self.ejerciciosSemanales:
            med = med + nota
        return med / len(self.ejerciciosSemanales)
```

[4]:
```python
# Creo una variable de la clase estudiante
# A la variable estu se le llama objeto
estu = Estudiante()
# Le doy valor a sus atributos
estu.nombre = "Pepe"
estu.nota = [10, 4, 6]
estu.ejerciciosSemanales = [4, 7, 8, 9, 3]
# Llamo al método e imprimo su valor
# Como el único parámetro que tiene es self, aquí lo llamo sin
# parámetros (no se le da valor a self al invocar un método)
# en este caso 'self' es 'estu'
print("La media es:", estu.media())
```

```
La media es: 6.2
```

3.1 Método init

- Es uno de los métodos *especiales* o *mágicos* de Python

- Se debe llamar `__init__`, si se llama de otra forma Python no lo encuentra y no funciona
- Se utiliza para dar valor inicial a los atributos de un objeto cuando se crea el objeto y también para declarar los atributos
- Aunque se pueden declarar atributos fuera del init, como vimos en la clase Estudiante, lo recomendable es crearlos dentro el init, única y exclusivamente en él
- Dentro de él se declaran los atributos de la clase poniendo `self.atributo`
- Debe tener un parámetro para dar valor a cada uno de los atributos. Los parámetros no tienen por qué llamarse como los atributos, aunque es lo normal
- Aunque siempre es recomendable que las clases tengan un método init, su verdadera potencia se ve cuando hay que hacer cálculos

[5]:
```
class Fecha:
    """ Clase que representa a una fecha con día, mes, año y si es
    año bisiesto. Contiene un método init para declarar e inicializar
    los atributos"""

    def __init__(self, dia: int, mes: str, anio: int, bis: bool):
        """ Método init que declara e inicializa los atributos"""
        # Todo lo que pone self.algo es un atributo
        # Damos valor a los atributos usando los parámetros
        self.dia = dia
        self.mes = mes
        self.anio = anio
        # Nótese que el parámetro y el atributo no tienen por qué
        # tener el mismo nombre
        self.bisiesto = bis
```

[6]:
```
# Ahora para crear objetos tengo que dar valor a cada uno de los
# parámetros
mi_fecha = Fecha(25, "noviembre", 2020, True)
print(mi_fecha.mes)
otra_fecha = Fecha(26, "noviembre", 2020, True)
print(otra_fecha.dia)
# Pero una vez creado el objeto puedo cambiar el valor del atributo
# como hacía antes
otra_fecha.mes = "octubre"
print(otra_fecha.mes)
```

```
noviembre
26
octubre
```

[7]:
```
class Fecha:
    """ Clase que representa a una fecha con día, mes, año y si es
    año bisiesto. Contiene un método init para declarar e inicializar
    los atributos. Calcula el valor del atributo 'bisiesto' sin
    necesidad de recibirlo como parámetro"""
```

```
    def __init__(self, dia: int, mes: str, anio: int):
        # Todo lo que pone self.algo es un atributo
        # Para los tres primeros atributos usamos los parámetros
        self.dia = dia
        self.mes = mes
        self.anio = anio
        # bisiesto lo calculamos
        if (self.anio % 4 == 0 and (self.anio % 100 != 0 or
                                    self.anio % 400 == 0)):
            self.bisiesto = True
        else:
            self.bisiesto = False
```

[8]:
```
# Ahora creamos el objeto con solo 3 parámetros
mi_fecha = Fecha(25, "noviembre", 2020)
mi_fecha.dia = 28
print("El año es", mi_fecha.anio)
print("¿Es bisiesto?", mi_fecha.bisiesto)
# Si imprimimos el objeto obtenemos fichero.Clase dirección de memoria
print(mi_fecha)
```

```
El año es 2020
¿Es bisiesto? True
<__main__.Fecha object at 0x05F9E270>
```

[9]:
```
class Fecha:
    """ Clase que representa a una fecha con día, mes, año y si es
    año bisiesto. Calcula el valor de bisiesto sin necesidad de
    recibirlo. Comprueba si los valores recibidos son correctos"""

    def __init__(self, dia: int, mes: str, anio: int):
        # Todo lo que pone self.algo es un atributo
        # Compruebo que el año es correcto
        if anio != 0:
            self.anio = anio
        else:
            # Si no lo es, le doy un valor seguro
            self.anio = 1900
        # Compruebo que el mes es correcto
        # Creo un diccionario con los meses y sus días
        # Esta es una variable local, que no lleva self, porque no
        # quiero que sea un atributo del objeto. Cuando termina el
        # método init, este diccionario desaparece
        dias_mes = {'enero': 31, 'febrero': 28, 'marzo': 31, 'abril': 30,
                    'mayo': 31, 'junio': 30, 'julio': 31, 'agosto': 31,
                    'septiembre': 30, 'octubre': 31, 'noviembre':30,
```

```
                    'diciembre': 31}
        # Si el mes que me han dado, cambiado a minúsculas, es una de las
        # claves del diccionario anterior, es un mes correcto
        if mes.lower() in dias_mes:
            # Entonces lo guardo
            self.mes = mes.lower()
        else:
            # Si no, pongo Enero
            self.mes = 'enero'
        # Calculo el valor de bisiesto
        if (self.anio % 4 == 0 and (self.anio % 100 != 0 or
                                    self.anio % 400 == 0)):
            self.bisiesto = True
            # Si es bisiesto cambio los días de Febrero
            dias_mes['febrero'] = 29
        else:
            self.bisiesto = False
        # Compruebo si el día es correcto
        if dia > 0 and dia <= dias_mes[self.mes]:
            self.dia = dia
        else:
            self.dia = 1
```

[10]:
```
# Si intento dar valores no correctos a los atributos, me pone unos
# valores por defecto
fech = Fecha(33, "del mes que me da la gana", 0)
print(fech.dia)
print(fech.mes)
print(fech.anio)
print(fech.bisiesto)
```

```
1
enero
1900
False
```

[11]:
```
class Fecha:
    """ Clase que representa a una fecha con día, mes, año y si es
    año bisiesto. Calcula el valor de bisiesto sin necesidad de
    recibirlo. Comprueba si los valores recibidos son correctos.
    Tiene valores por defecto en los parámetros"""

    def __init__(self, dia: int = 1, mes: str = "enero",
                 anio: int = 1900):
        # Todo lo que pone self.algo es un atributo
        # Compruebo que el año es correcto
        if anio != 0:
```

```
            self.anio = anio
        else:
            # Si no lo es, le doy un valor seguro
            self.anio = 1900
        # Compruebo que el mes es correcto
        # Creo un diccionario con los meses y sus días
        # Esta es una variable local, que no lleva self, porque no
        # quiero que sea un atributo del objeto. Cuando termina el
        # método init, este diccionario desaparece
        dias_mes = {'enero': 31, 'febrero': 28, 'marzo': 31, 'abril': 30,
                    'mayo': 31, 'junio': 30, 'julio': 31, 'agosto': 31,
                    'septiembre': 30, 'octubre': 31, 'noviembre':30,
                    'diciembre': 31}
        # Si el mes que me han dado, cambiado, a minúsculas es una de las
        # claves del diccionario anterior, es un mes correcto
        if mes.lower() in dias_mes:
            # Entonces lo guardo
            self.mes = mes.lower()
        else:
            # Si no, pongo Enero
            self.mes = 'enero'
        # Calculo el valor de bisiesto
        if (self.anio % 4 == 0 and (self.anio % 100 != 0 or
                                    self.anio % 400 == 0)):
            self.bisiesto = True
            # Si es bisiesto cambio los días de Febrero
            dias_mes['febrero'] = 29
        else:
            self.bisiesto = False
        # Compruebo si el día es correcto
        if dia > 0 and dia <= dias_mes[self.mes]:
            self.dia = dia
        else:
            self.dia = 1
```

[12]:
```
# Creo un objeto dando valor a todos los parámteros
mf = Fecha(3, "enero", 2021)
# También puedo crearlo sin darle valor a ninguno
mf2 = Fecha()
print(mf2.anio)
# O dando valor solamente a algunos por su nombre
mf3 = Fecha(mes = "marzo")
print(mf3.mes)
# O por su orden
mf4 = Fecha(22, "enero")
print(mf4.anio)
```

```
1900
marzo
1900
```

3.2 Más métodos especiales

- Python tiene muchos métodos especiales. Todos tienen una estructura de nombre común: `__nombre__`
- Podemos implementarlos para que nuestros objetos sean más potentes y más versátiles
- Recordad que nunca se debe crear una variable nueva con `self` si no es en el `init`. Si me hacen falta variables auxiliares en estos métodos irán siempre sin `self` (tampoco es buena idea crearlas en el `init` con `self` si solo las voy a usar en estos métodos)

3.2.1 Método `__str__(self)`

- Como hemos visto, si imprimo un objeto obtengo `nombre_fichero.nombre_clase dirección_de_memoria`
- Si quiero modificarlo tengo que implementar el método `__str__(self)`
- Tiene que llamarse así, no se puede cambiar el nombre **ni añadir ningún parámetro**

3.2.2 Método `__eq__(self, otro_objeto)`

- Si comparo dos objetos con `==` me dice que son distintos aunque los atributos sean iguales. Para cambiarlo implemento el método `__eq__(self, otro_objeto)`
- Recibe dos parámetros, `self` que representa a este objeto y `otro_objeto` que representa al objeto con el que estamos comparando

[13]:
```python
class Fecha:
    """ Clase que representa a una fecha con día, mes, año y si es
    año bisiesto. Calcula el valor de bisiesto sin necesidad de
    recibirlo. Comprueba si los valores recibidos son correctos.
    Tiene valores por defecto en los parámetros"""

    def __init__(self, dia: int = 1, mes: str = "enero",
                 anio: int = 1900):
         # Todo lo que pone self.algo es un atributo
        # Compruebo que el año es correcto
        if anio != 0:
            self.anio = anio
        else:
            # Si no lo es, le doy un valor seguro
            self.anio = 1900
        # Compruebo que el mes es correcto
        # Creo un diccionario con los meses y sus días
        # Esta es una variable local, que no lleva self, porque no
        # quiero que sea un atributo del objeto. Cuando termina el
        # método init, este diccionario desaparece
        dias_mes = {'enero': 31, 'febrero': 28, 'marzo': 31, 'abril': 30,
                    'mayo': 31, 'junio': 30, 'julio': 31, 'agosto': 31,
```

```
                'septiembre': 30, 'octubre': 31, 'noviembre':30,
                'diciembre': 31}
        # Si el mes que me han dado, cambiado a minúsculas, es una de las
        # claves del diccionario anterior, es un mes correcto
        if mes.lower() in dias_mes:
            # Entonces lo guardo
            self.mes = mes.lower()
        else:
            # Si no, pongo Enero
            self.mes = 'enero'
        # Calculo el valor de bisiesto
        if (self.anio % 4 == 0 and (self.anio % 100 != 0 or
                                    self.anio % 400 == 0)):
            self.bisiesto = True
            # Si es bisiesto cambio los días de Febrero
            dias_mes['febrero'] = 29
        else:
            self.bisiesto = False
        # Compruebo si el día es correcto
        if dia > 0 and dia <= dias_mes[self.mes]:
            self.dia = dia
        else:
            self.dia = 1

    def __str__(self):
        """Si este método existe, Python lo ejecuta cuando imprimo
        un objeto de esta clase. Debe devolver la cadena que yo
        quiero que se imprima"""
        if self.bisiesto:
            # bis es una variable local que uso aquí y que no quiero
            # que sobreviva cuando el método acaba
            bis = " bisiesto"
        else:
            bis = " no bisiesto"
        # resultado es también una variable local
        resultado = (str(self.dia) + " de " + self.mes + " de " +
                     str(self.anio) + bis)
        return resultado

    # El primer parámetro representa a este objeto, el segundo
    # parámetro al objeto con el que quiero compararlo
    def __eq__(self, otro):
        """ Este método se llama cuando uso == para comparar dos
        objetos """
        # En este caso en lugar de crear una variable para guardar
        # el resultado y luego devolver su valor, hago el cálculo y
        # lo devuelvo. También se podría hacer de la otra forma
```

```
        return (self.dia == otro.dia and self.mes == otro.mes and
                self.anio == otro.anio)
```

[14]:
```
mf5 = Fecha(22, "enero", 1946)
# Al tener método str puedo imprimir el objeto
print(mf5)
mf6 = Fecha(22, "enero", 1946)
# Si el método __eq__ no estuviera implementado, esto daría False
print(mf6 == mf5)
print(mf5 == mf6)
# También puedo llamar al método __eq__ directamente en lugar de
# usar == pero no se suele hacer, es más cómodo usar ==
print(mf6.__eq__(mf5))
```

```
22 de enero de 1946 no bisiesto
True
True
True
```

4 Composición

- Una clase que tiene un atributo que es un objeto de otra clase

[15]:
```
class Persona:
    """ Clase que representa a una persona, uno de sus atributos
    es un objeto de la clase Fecha"""

    def __init__(self, nombre: str, apellidos: str, nacim: Fecha):
        ''' Método init, recibimos un valor para cada atributo'''
        self.nombre = nombre
        self.apellidos = apellidos
        self.nacimiento = nacim
```

[16]:
```
# Creo un objeto de la clase Persona
# Tengo dos formas de hacerlo
# Usando una variable auxiliar para la fecha
f = Fecha(12, "marzo", 2000)
p = Persona("Pepe", "Perez Gomez", f)
# Para ver el valor de un atributo normal uso objeto.atributo como
# de costumbre
print(p.nombre)
# Pero para acceder a uno de los valores de un atributo que es un
# objeto tengo que usar objeto.atributo.atributo
print(p.nacimiento.dia)
# Si imprimo p.nacimiento, me llamará al str de Fecha. Si no estuviera
# creado pondría lo de nombre de clase y dirección de memoria
print(p.nacimiento)
```

```python
# Para cambiar el día de nacimiento
p.nacimiento.dia = 4
print(p.nacimiento)
# Ojo porque p.nacimiento y f son el mismo objeto, cualquier cambio
# en uno se refleja en el otro. Esto es porque los objetos son
# mutables por lo que es como si pasaran por referencia
print(f)

# Segunda forma de crear el objeto: sin usar variable auxiliar
p2 = Persona("Maria", "Gomez Kemp", Fecha(12, "febrero", 2002))
print(p2.nacimiento)

# Para imprimir el nacimiento sin bisiesto tengo que imprimir a mano
print(p2.nacimiento.dia, "de", p2.nacimiento.mes, "de",
     p2.nacimiento.anio)
# Aunque también podría usar lo que me devuelve str y modificarlo
cadena = p2.nacimiento.__str__()
if "no bisiesto" in cadena:
    pos_bisiesto = cadena.index("no bisiesto")
else:
    pos_bisiesto = cadena.index("bisiesto")
print(cadena[0:pos_bisiesto])
```

```
Pepe
12
12 de marzo de 2000 bisiesto
4 de marzo de 2000 bisiesto
4 de marzo de 2000 bisiesto
12 de febrero de 2002 no bisiesto
12 de febrero de 2002
12 de febrero de 2002
```

5 Atributos y métodos privados

- Los atributos tienen dos características
  - Son accesibles por todos los métodos de la clase
  - Son accesibles desde el exterior de la clase
- ¿Cómo hago si quiero conseguir la primera pero no la segunda?: Uso atributos privados
- Un atributo privado es un atributo que no se puede ver ni cambiar fuera de la clase
- La forma de hacer un atributo privado es empezar su nombre con dos guiones bajos: `self.__atributo` (no puede acabar con `__`, entonces no sería privado)
- Los atributos privados permiten *ocultación de la información*: oculto cómo se guarda la información internamente en la clase
- Reglas para decidir cómo guardar una variable en una clase:
  - Si quiero que la variable se vea en todos los métodos de la clase y además también fuera: atributo normal (público)

    - Si quiero que la variable se vea en todos los métodos de la clase pero no fuera: atributo privado
    - Si solo lo voy a usar en un método: variable local
- Debería reducir el número de atributos públicos al máximo
- También puedo tener métodos privados `def __nombre(self,...)`: métodos que voy a usar internamente pero que no son visibles fuera

[17]:
```python
class Fecha:
    """ Clase que representa a una fecha con día, mes, año y si es
    año bisiesto. Calcula el valor de bisiesto sin necesidad de
    recibirlo. Comprueba si los valores recibidos son correctos.
    Tiene valores por defecto en los parámetros y un atributo privado"""

    def __init__(self, dia: int = 1, mes: str = "enero",
                 anio: int = 1900):
         # Todo lo que pone self.algo es un atributo
        # Compruebo que el año es correcto
        if anio != 0:
            self.anio = anio
        else:
            # Si no lo es, le doy un valor seguro
            self.anio = 1900
        # Compruebo que el mes es correcto
        # Creo un diccionario con los meses y sus días. Como lo voy a
        # usar en otro método y no quiero que se vea fuera de la clase,
        # lo creo como atributo privado
        self.__dias_mes = {'enero': 31, 'febrero': 28, 'marzo': 31, 'abril': 30,
                   'mayo': 31, 'junio': 30, 'julio': 31, 'agosto': 31,
                   'septiembre': 30, 'octubre': 31, 'noviembre':30,
                   'diciembre': 31}
        # Si el mes que me han dado cambiado a minúsculas es una de las
        # claves del diccionario anterior, es un mes correcto
        if mes.lower() in self.__dias_mes:
            # Entonces lo guardo
            self.mes = mes.lower()
        else:
            # Si no, pongo Enero
            self.mes = 'enero'
        # Calculo el valor de bisiesto
        if (self.anio % 4 == 0 and (self.anio % 100 != 0 or
                                    self.anio % 400 == 0)):
            self.bisiesto = True
            # Si es bisiesto cambio los días de Febrero
            self.__dias_mes['febrero'] = 29
        else:
            self.bisiesto = False
        # Compruebo si el día es correcto
```

```
        if dia > 0 and dia <= self.__dias_mes[self.mes]:
            self.dia = dia
        else:
            self.dia = 1

    def __str__(self):
        """Si este método existe, Python lo ejecuta cuando imprimo
        un objeto de esta clase. Debe devolver la cadena que yo
        quiero que se imprima"""
        if self.bisiesto:
            # bis es una variable local que uso aquí y que no quiero
            # que sobreviva cuando el método acaba
            bis = " bisiesto"
        else:
            bis = " no bisiesto"
        # resultado es también una variable local
        resultado = (str(self.dia) + " de " + self.mes + " de " +
                     str(self.anio) + bis)
        return resultado

    # El primer parámetro representa a este objeto, el segundo
    # parámetro al objeto con el que quiero compararlo
    def __eq__(self, otro):
        """ Este método se llama cuando uso == para comparar dos
        objetos """
        # En este caso en lugar de crear una variable para guardar
        # el resultado y luego devolver su valor, hago el cálculo y
        # lo devuelvo. También se podría hacer de la otra forma
        return (self.dia == otro.dia and self.mes == otro.mes and
                self.anio == otro.anio)

    def dias_en_mes(self, mes: str):
        """ Devuelve los días que hay en un mes"""
        if mes.lower() in self.__dias_mes:
            return self.__dias_mes[mes.lower()]
        else:
            return None
```

[18]:
```
d = Fecha(12, "Diciembre", 2020)
print(d.dias_en_mes("junio"))
# Al hacer dias_mes privado ya no puedo verlo fuera, ni para leerlo
# ni para cambiarlo. Estas líneas darían error
# print(d.dias_mes)
# d.dias_mes["marzo"] = 432
# print(d.dias_mes)
# print(d.__dias_mes)
```

30

6 Propiedades

- Con el método init puedo/debo comprobar que los valores que me dan para los parámetros son adecuados y si no hacer algo al respecto. Pero no me vale para comprobar que los valores son correctos una vez que el objeto se ha creado
- Todos los lenguajes de programación orientados a objetos proporcionan algún mecanismo para evitar dar valores incorrectos a los atributos cuando el objeto ya está creado. En casi todos ellos se consigue mediante una mezcla de atributos privados y unos métodos especiales que se llaman `getters` y `setters`
- Python lo resuelve por medio de una forma especial: el uso de *propiedades*
- Pasos para usar propiedades:

1. Declaro los atributos en el método init, sin realizar ningún tipo de comprobación
2. Para cada atributo cuyo valor quiera controlar voy a crear dos métodos. El primero se debe llamar como el atributo (`def atributo(self):`) y no recibir ningún parámetro, aparte del self. Además se debe decorar con `@property`, que es la palabra clave que le indica a Python que esto no es un método normal sino una propiedad.
3. Dentro de este método propiedad, debo devolver el valor del atributo pero **como si el atributo fuera privado**
4. A continuación hago un segundo método que también se llama como el atributo y que recibe un parámetro, aparte del self, `def atributo(self, valor)` y que debe estar decorado con `@atributo.setter`. Aquí también tengo que tratar al atributo como si fuera privado (en el resto de métodos de la clase lo seguiré poniendo como público). En este método es donde voy a recibir el valor del atributo y voy a hacer las comprobaciones necesarias.

[19]:
```
# El init me permite evitar valores erróneos
f = Fecha(222, "mes inexistente", 0)
print(f)
# Pero una vez creado el objeto ya no me sirve
f.dia = 222
print(f)
```

```
1 de enero de 1900 no bisiesto
222 de enero de 1900 no bisiesto
```

[20]:
```
class Fecha:
    """Clase que representa una fecha y usa propiedades para evitar
    valores erróneos en los atributos"""
    def __init__(self, dia: int = 1, mes: str = "enero",
                 anio: int = 1900):
        """Método init"""
        # Atributo privado
        self.__dias_mes = {'enero': 31, 'febrero': 28, 'marzo': 31, 'abril': 30,
                   'mayo': 31, 'junio': 30, 'julio': 31, 'agosto': 31,
                   'septiembre': 30, 'octubre': 31, 'noviembre':30,
                   'diciembre': 31}
```

```
        # No hago comprobaciones sobre valores en el init
        self.anio = anio
        self.mes = mes
        self.dia = dia
        # No declaro un atributo para bisiesto porque voy a poder
        # calcularlo

    # Aquí estoy definiendo el atributo anio como propiedad
    @property
    def anio(self) -> int:
        """ Este es el método que se va a usar para leer el valor
        del atributo"""
        # Aquí debo devolver el valor del atributo como si fuera
        # privado
        return self.__anio

    @anio.setter
    def anio(self, anio: int):
        """ Este es el método que se va a utilizar para dar valor al
        atributo """
        # Aquí también tengo que considerar al atributo como si fuera
        # privado
        if anio != 0:
            self.__anio = anio
        else:
            # Si no lo es, le doy un valor seguro
            self.__anio = 1900

        # Cada vez que cambie el año debo comprobar si es bisiesto
        # para cambiar los días de Febrero
        # Como ya le he dado valor, ya no hace falta que lo trate
        # como privado
        if (self.anio % 4 == 0 and (self.anio % 100 != 0 or
                                    self.anio % 400 == 0)):
            # Si es bisiesto cambio los días de Febrero a 29
            self.__dias_mes['febrero'] = 29
        else:
            # Si no es bisiesto cambio los días de Febrero a 28
            self.__dias_mes['febrero'] = 28

    @property
    # Propiedad sin setter
    def bisiesto(self) -> bool:
        """ Para los atributos de solo lectura, creo una propiedad
        pero no un setter. Así el usuario puede ver lo que vale
        bisiesto pero no cambiarlo"""
        return (self.anio % 4 == 0 and self.anio % 100 != 0 or
```

```
                                    self.anio % 400 == 0)

    # Propiedades y setters para mes y dia
    @property
    def mes(self) -> str:
        return self.__mes

    @mes.setter
    def mes(self, mes: str):
        if mes.lower() in self.__dias_mes:
            self.__mes = mes.lower()
        else:
            self.__mes = "enero"

    @property
    def dia(self) -> int:
        return self.__dia

    @dia.setter
    def dia(self, dia: int):
        if dia > 0 and dia <= self.__dias_mes[self.mes]:
            self.__dia = dia
        else:
            self.__dia = 1

    def __str__(self) -> str:
        """Si este método existe, Python lo ejecuta cuando imprimo
        un objeto de esta clase. Debe devolver la cadena que yo
        quiero que se imprima"""
        # Aunque en ningún sitio declaré self.bisiesto, al haber hecho
        # la propiedad es como si existiese (solo para leerlo)
        if self.bisiesto:
            # bis es una variable local que uso aquí y que no quiero
            # que sobreviva cuando el método acaba
            bis = " bisiesto"
        else:
            bis = " no bisiesto"
        # resultado es también una variable local
        resultado = (str(self.dia) + " de " + self.mes + " de " +
                     str(self.anio) + bis)
        return resultado
```

[21]:
```
# Pongo una fecha incorrecta al crear el objeto
f = Fecha(1, "juenero", 0)
# Me la corrige
print(f)
# Intento dar valores incorrectos a los atributos y me los corrige
```

```
f.anio = 0
print(f)
f.mes = "juenero"
print(f)
# Imprimo el valor de bisiesto
print(f.bisiesto)
# Pero si intento darle valor me salta un error
f.bisiesto = False
```

```
1 de enero de 1900 no bisiesto
1 de enero de 1900 no bisiesto
1 de enero de 1900 no bisiesto
False
```

```
      ␣
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-21-af74d38f622d> in <module>
        11 print(f.bisiesto)
        12 # Pero si intento darle valor me salta un error
   ---> 13 f.bisiesto = False


       AttributeError: can't set attribute
```

7 Lanzamiento de excepciones

- Cuando me dan un valor incorrecto para un atributo, en lugar de dar yo un valor seguro muchas veces es mejor lanzar un error (una excepción)
- Si el valor es incorrecto debería lanzar un ValueError: `raise ValueError(mensaje)` el mensaje es opcional
- Si lo que es incorrecto es el tipo lanzo un TypeError: `raise TypeError(mensaje)`
- La excepción hace que el programa se pare, pero puedo usar las instrucciones `try ... except` para que el programa gestione la excepción y no falle (no lo veremos este curso)
- Decicir si debo lanzar una excepción o poner un valor por defecto es parte del diseño de mi clase y debería estar en la documentación de mi programa

[23]:
```
class Fecha:
    """Clase que representa una fecha y usa propiedades para evitar
    valores erróneos en los atributos. Si algún valor es erróneo
    lanza una excepción o pone otro valor por defecto"""
    def __init__(self, dia: int = 1, mes: str = "enero",
```

```
                anio: int = 1900):
        """Método init"""
        # Atributo privado
        self.__dias_mes = {'enero': 31, 'febrero': 28, 'marzo': 31, 'abril': 30,
                   'mayo': 31, 'junio': 30, 'julio': 31, 'agosto': 31,
                   'septiembre': 30, 'octubre': 31, 'noviembre':30,
                   'diciembre': 31}
        # No hago comprobaciones sobre valores en el init
        self.anio = anio
        self.mes = mes
        self.dia = dia
        # No declaro un atributo para bisiesto porque voy a poder
        # calcularlo

    # Aquí estoy definiendo el atributo anio como propiedad
    @property
    def anio(self) -> int:
        """ Este es el método que se va a usar para leer el valor
        del atributo"""
        # Aquí debo devolver el valor del atributo como si fuera
        # privado
        return self.__anio

    @anio.setter
    def anio(self, anio: int):
        """ Este es el método que se va a utilizar para dar valor al
        atributo. Si el tipo o el valor son incorrectos lanza una excepción"""
        # Compruebo que el año es entero
        if type(anio) != int:
            # Si no lo es, lanzo una excepción
            raise TypeError("El año debe ser un entero")
        # Aquí también tengo que considerar al atributo como si fuera
        # privado
        elif anio != 0:
            self.__anio = anio
        else:
            # Si el año es 0 también lanzo una excepción
            raise ValueError("0 no es un valor correcto para el año")

        # Si se ha lanzado una excepción, este código ya no se ejecuta
        # Cada vez que cambie el año debo comprobar si es bisiesto
        # para cambiar los días de Febrero
        # Como ya le he dado valor, ya no hace falta que lo trate
        # como privado
        if (self.anio % 4 == 0 and (self.anio % 100 != 0 or
                                    self.anio % 400 == 0)):
            # Si es bisiesto cambio los días de Febrero a 29
```

```
            self.__dias_mes['febrero'] = 29
        else:
            # Si no es bisiesto cambio los días de Febrero a 28
            self.__dias_mes['febrero'] = 28

    @property
    # Propiedad sin setter
    def bisiesto(self) -> bool:
        """ Para los atributos de solo lectura, creo una propiedad
        pero no un setter. Así el usuario puede ver lo que vale
        bisiesto pero no cambiarlo"""
        return (self.anio % 4 == 0 and self.anio % 100 != 0 or
                                    self.anio % 400 == 0)

    # Propiedades y setters para mes y dia
    @property
    def mes(self) -> str:
        return self.__mes

    @mes.setter
    def mes(self, mes: str):
        """ Asigna valor al mes, si el nombre del mes no es correcto
        o no es una cadena de texto, lanza una excepción"""
        if type(mes) != str:
            # Si no es un str lanzo excepción
            raise TypeError("El valor del mes debe ser su nombre")
        elif mes.lower() in self.__dias_mes:
            self.__mes = mes.lower()
        else:
            # Si el nombre no es correcto lanzo una excepción
            raise ValueError("%s no es un mes válido. Los valores " +
                             "válidos son %s" %(mes, self.__dias_mes.keys()))

    @property
    def dia(self) -> int:
        return self.__dia

    @dia.setter
    def dia(self, dia: int):
        """ Asigna valor al día. Si no es un entero lanza una excepción.
        Si no es un valor correcto asigna 1 o el número máximo de días del␣
↪mes"""
        if type(dia) != int:
            raise TypeError("El día debe ser entero")
        # En caso de que el día no sea correcto pondremos 1 o el
        # máximo de ese mes en lugar de lanzar la excepción
        elif dia <= 0:
```

```
            self.__dia = 1
        elif dia > self.__dias_mes[self.mes]:
            self.__dia = self.__dias_mes[self.mes]
        else:
            self.__dia = dia

    def __str__(self) -> str:
        """Si este método existe, Python lo ejecuta cuando imprimo
        un objeto de esta clase. Debe devolver la cadena que yo
        quiero que se imprima"""
        # Aunque en ningún sitio declaré self.bisiesto, al haber hecho
        # la propiedad es como si existiese (solo para leerlo)
        if self.bisiesto:
            # bis es una variable local que uso aquí y que no quiero
            # que sobreviva cuando el método acaba
            bis = " bisiesto"
        else:
            bis = " no bisiesto"
        # resultado es también una variable local
        resultado = (str(self.dia) + " de " + self.mes + " de " +
                     str(self.anio) + bis)
        return resultado
```

[24]:
```
# Fecha con el año mal, salta una excepción
f = Fecha(1, "enero", 'pepe')
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-24-dcb2f1808f79> in <module>
  1 # Fecha con el año mal, salta una excepción
----> 2 f = Fecha(1, "enero", 'pepe')


<ipython-input-23-eac45c83171a> in __init__(self, dia, mes, anio)
 12                     'diciembre': 31}
 13         # No hago comprobaciones sobre valores en el init
---> 14         self.anio = anio
 15         self.mes = mes
 16         self.dia = dia


<ipython-input-23-eac45c83171a> in anio(self, anio)
```

```
        34         if type(anio) != int:
        35             # Si no lo es, lanzo una excepción
---->   36             raise TypeError("El año debe ser un entero")
        37         # Aquí también tengo que considerar al atributo como si fuera
        38         # privado


TypeError: El año debe ser un entero
```

[25]:
```python
# Fecha con el dia mal, pongo valor por defecto
f = Fecha(44, "enero", 2021)
print(f)
```

```
31 de enero de 2021 no bisiesto
```

8 Características de la programación orientada a objetos

- Las tres características principales de la OO son *Encapsulación*, *Herencia* y *Polimorfismo*
- Encapsulación: ponemos juntos los datos (atributos) y las funciones (métodos) que trabajan con esos datos. No exclusivo de orientación a objetos, también se hace a menudo en programación estructurada.
- En POO la encapsulación incluye ocultación de la información (atributos privados y métodos para cambiar y leer el valor de los atributos: properties y setters en el caso de Python) lo que la hace aún más potente

9 Herencia

- Forma exclusiva de la POO de crear una clase a partir de otra clase
- Se utiliza cuando quiero crear una clase nueva partiendo de una clase ya existente
- Es equivalente a copiar el código de la clase antigua en la nueva y realizar las modificaciones y adiciones pertinentes, pero sin necesidad de copiarlo
- La clase base se llama clase madre o super-clase. La clase nueva se denomina clase hija o subclase
- Si la clase B hereda de la clase A, incluye todos sus métodos y atributos
- Ventajas: no tengo que copiar el código y sobre todo: cualquier cambio en la clase madre se refleja automáticamente en la clase hija

[26]:
```python
class Producto:
    """ Clase que representa un producto, será la clase madre de
    otra clase"""
    def __init__(self, precio: float, nombre: str):
        self.precio = precio
        self.nombre = nombre

    @property
    def precio(self) -> float:
```

```
        return self.__precio

    @precio.setter
    def precio(self, valor: float):
        if type(valor) != float and type(valor) != int:
            raise TypeError("El precio debe ser un número")
        elif valor < 0:
            self.__precio = 0
        else:
            self.__precio = valor

    def __str__(self):
        return self.nombre +": "+str(self.precio)
```

[27]:
```
p = Producto(3.5, "bolígrafo")
print(p)
```

```
bolígrafo: 3.5
```

[28]:
```
class ProductoFresco:
    """ Ejemplo de una clase producto fresco sin herencia. Copio
    todo el código de la otra clase y lo modifico"""
    def __init__(self, precio: float, nombre: str, caducidad: Fecha):
        self.precio = precio
        self.nombre = nombre
        self.caducidad = caducidad

    @property
    def precio(self) -> float:
        return self.__precio

    @precio.setter
    def precio(self, valor: float):
        if type(valor) != float and type(valor) != int:
            raise TypeError("El precio debe ser un número")
        elif valor < 0:
            self.__precio = 0
        else:
            self.__precio = valor

    def __str__(self):
        return (self.nombre +": "+str(self.precio)+" caduca el "
                + str(self.caducidad))
```

[29]:
```
pf = ProductoFresco(2.4,"naranjas", Fecha(1,"enero", 2021))
print(pf)
```

```
naranjas: 2.4 caduca el 1 de enero de 2021 no bisiesto
```

[33]:
```python
class ProductoFresco2(Producto):
    """ Ejemplo de clase que usa herencia. Solo hay que incluir un
    init para definir los atributos"""
    def __init__(self, precio: float, nombre: str, caducidad: Fecha):
        self.precio = precio
        self.nombre = nombre
        self.caducidad = caducidad
    # El resto de métodos, incluidos properties y setters están aunque
    # no los hayamos copiado
```

[31]:
```python
pf2 = ProductoFresco2(2.4,"zanahorias", Fecha(11,"enero", 2021))
# Aquí estamos llamando al str, pero es el de Producto, por lo que
# no incluye la caducidad
print(pf2)
# Pero se puede ver que las properties funcionan
pf2.precio = -12.0
print(pf2)
```

```
zanahorias: 2.4
zanahorias: 0
```

[32]:
```python
# Añadimos un nuevo método a Producto y vemos que sin hacer nada
# también existe en ProductoFresco2
class Producto:
    """ Clase que representa un producto, será la clase madre de
    otra clase"""
    def __init__(self, precio: float, nombre: str):
        self.precio = precio
        self.nombre = nombre

    @property
    def precio(self) -> float:
        return self.__precio

    @precio.setter
    def precio(self, valor: float):
        if type(valor) != float and type(valor) != int:
            raise TypeError("El precio debe ser un número")
        elif valor < 0:
            self.__precio = 0
        else:
            self.__precio = valor

    def __str__(self):
        return self.nombre +": "+str(self.precio)
```

```
    def rebajar(self):
        """ Pone el producto al 50% """
        self.precio /= 2
```

[34]:
```
pf2 = ProductoFresco2(2.4,"zanahorias", Fecha(11,"enero", 2021))
pf2.rebajar()
print(pf2)
```

```
zanahorias: 1.2
```

[35]:
```
# Si los métodos heredados no son suficientes, podemos sobrescribirlos
# reutilizando si es necesario el de la clase madre mediante super()
class ProductoFresco2(Producto):
    # Ya que en Producto había un init para inicializar sus atributos
    # aquí lo llamaremos, y luego inicializaremos los atributos nuevos
    def __init__(self, precio: float, nombre: str, caducidad: Fecha):
        # super().metodo(parametros) significa: ve a la clase madre, busca
        # el método y ejecútalo
        # No puedo llamar directamente al init de la clase madre porque lo
        # he reescrito
        super().__init__(precio, nombre)
        self.caducidad = caducidad

    # Como el str de producto, no muestra toda la información que
    # yo necesitaría, tengo sobrescribirlo
    def __str__(self):
        # Puedo llamar al de la clase madre y completarlo
        return (super().__str__() + " caduca el "
                + str(self.caducidad))

    def mega_rebaja(self):
        """ Pone el artículo al 25%, llamando 2 veces a rebajar()"""
        # Como no he sobrescrito el método rebajar, lo puedo llamar
        # directamente, sin poner super()
        self.rebajar()
        # aunque también podría usar super().rebajar()
        super().rebajar()
```

[36]:
```
pf3 = ProductoFresco2(1.2, "berzas", Fecha(12,"diciembre",2020))
print(pf3)
pf3.mega_rebaja()
print(pf3)
```

```
berzas: 1.2 caduca el 12 de diciembre de 2020 bisiesto
berzas: 0.3 caduca el 12 de diciembre de 2020 bisiesto
```

[37]:
```python
class Mueble():
    """ Otro ejemplo de herencia. Clase madre. """
    def __init__(self,nombre:str,precio:float,material:str):
        self.nombre=nombre
        self.precio=precio
        self.material=material

    @property
    def nombre(self):
        return self.__nombre
    @nombre.setter
    def nombre(self, nombre):
        self.__nombre = nombre
    @property
    def precio(self):
        return self.__precio
    @precio.setter
    def precio(self, precio):
        if type(precio) != float:
            raise TypeError("Precio debe ser un número")
        elif precio < 0:
            raise ValueError("El precio debe ser positivo")
        else:
            self.__precio = precio
    @property
    def material(self):
        return self.__material
    @material.setter
    def material(self, material):
        self.__material = material

    def __str__(self):
        texto= ("Nombre: "+str(self.nombre)+"\n"+"Precio: "
        +str(self.precio)+"\n"+"Material: "+str(self.material))
        return texto

    def plastificar(self):
        """ Cambia material a plastico """
        self.material="plastico"
        return self.material

# Clase hija
class Silla(Mueble):
    """ Clase que hereda de mueble, el atributo nombre será 'silla'
    siempre. Además incluye un nuevo atributo n_patas"""
    def __init__(self,precio:float,material:str,n_patas:int):
        super().__init__("silla",precio,material)
```

```
        self.n_patas = n_patas

    @property
    def n_patas(self):
        return self.__n_patas

    @n_patas.setter
    def n_patas(self, patas):
        if type(patas) != int:
            raise TypeError("Patas debe ser un entero")
        elif patas <= 0:
            # Si es negativo ponemos 4
            self.__n_patas = 4
        else:
            self.__n_patas = patas

    def __str__(self):
        return (super().__str__() + "\nNúmero de patas: "
                +str(self.n_patas))

silla = Silla(100.0,"madera",4)
print(silla)
silla.plastificar()
print(silla)
```

```
Nombre: silla
Precio: 100.0
Material: madera
Número de patas: 4
Nombre: silla
Precio: 100.0
Material: plastico
Número de patas: 4
```

10 Polimorfismo

- Es la capacidad de un objeto de una clase hija de comportarse como un objeto de la clase madre
- Como en Python no hay tipos no es muy relevante, pero en otros lenguajes simplifica mucho la programación

[38]:
```
silla = Silla(100.0,"madera",4)
mue = Mueble("Mesa", 40.0, "plástico")
# Imprimimos los tipos
print(type(silla))
print(type(mue))
# Vemos que un objeto de la clase hija es también un objeto de la clase
```

```
# madre
print(isinstance(silla, Silla))
print(isinstance(mue, Mueble))
print(isinstance(silla, Mueble))
print(isinstance(mue, Silla))
```

```
<class '__main__.Silla'>
<class '__main__.Mueble'>
True
True
True
False
```