Tema 3:

Programación orientada a objetos en Python

***Objetivos del tema**: este tema detalla la implementación en Python de la programación orientada a objetos estudiada en el tema anterior.*

INDICE:

1 Clases y herencia en Python

Una clase es la "plantilla" que usamos para crear los objetos. Todos los objetos pertenecen a una determinada clase. Un objeto que se crea a partir de una clase se dice que es una instancia de esa clase. Las distintas clases tienen distintas relaciones de herencia entre sí: una clase puede derivarse de otra, en ese caso la clase derivada o clase hija hereda los métodos y variables de la clase de la que se deriva o clase padre. En Python todas las clases tienen como primer padre una misma clase: la clase Object.

Vamos a continuación a profundizar en todos estos conceptos y a explicar su sintaxis en Python.

1.1 Definición de una clase

La forma más general de definición de una clase en Python es:

```
class NombreClase(nombreClasePadre o nombreClasesPadre):
  def __init__(self,atrib1,atrib2, atrib3):
    self.atrib1 = atrib1  # public attribute
    self._atrib2 = atrib2  # protected attribute
    self.__atrib3 = atrib3  # private attribute
```

nombreClase es el nombre que le queramos dar a nuestra clase, nombreClasePadre es el nombre de la clase padre o de las clases padre, si se produce una herencia múltiple, de la cual hereda los métodos y variables.

Veamos un ejemplo de clase en Python:

```
class Animal:
  def __init__(self, age = "0", name = "Godzilla"):
    '''constructor of an animal
    :param age: age of the animal: By default -> 0
    :param name: name of the animal. By Default: Godzilla
    :returns an instance of the Animal Class (One animal)
    '''
    # Todos estos atributos son públicos
```

```
    self.age = age # public attribute
    self.name = name # public attribute
    self.vivo = True
    Animal.numAnimales +=1
  def saluda():
    print('Hola')

  def mostrarNombre(self):
    print(self.nombre)

  def mostrarEdad(self):
    print(self.edad)
```

Sobrecarga de métodos

Python admite lo que se llama sobrecarga de métodos: puede haber varios métodos con el mismo nombre, pero a los cuales se les pasan distintos parámetros. Según los parámetros que se le pasen, se invocará al método con los valores por defecto especificados. Por ejemplo, las funciones saluda y add están sobrecargadas y permiten la entrada de diferentes parámetros en número (saluda) y en tipos de datos para que haga diferentes operaciones (add).

La función saluda implementa un polimorfismo por sobrecarga de funciones. La función saluda puede recibir 0, 1 o 2 parámetros, es decir, existen 4 funciones saluda diferentes y se puede realizar mediante la indicación de valores por defecto en los parámetros de la función. Esto hace que esos parámetros sean opcionales .

La función add implementa un polimorfismo de coerción, ya que una misma operación (suma) puede producir diferentes resultados dependiendo del tipo de operandos (sumar si los parámetros son enteros o concatenar si los parámetros son strings).

El polimorfismo por sustitución lo estudiaremos más adelante, en la sección 1.3.

En Python, al tener un tipado dinámico y utilizar el duck typing, no es necesario ni posible implementar el polimorfismo parámetrico utilizando genéricos.

```
class Animal:

  def __init__(self, age, name):
    self.age = age # public attribute
    self.name = name # public attribute
```

```
  def saluda(self, saludo='Hola', receptor = 'nuevo amigo'):
    print(saludo + " " + receptor)

  @staticmethod
  def add(a, b):
    if isinstance(a, int) and isinstance(b, int):
        return a + b
    elif isinstance(a, str) and isinstance(b, str):
        return " ".join((a, b))
    else:
        raise TypeError

  def mostrarNombre(self):
    print(self.nombre)

  def mostrarEdad(self):
    print(self.edad)
```

Constructores (__init__)

Constructores son métodos cuyo nombre es siempre __init__(self, [PARAM_OPCIONALES]) y que nunca devuelven ningún tipo de dato. Los constructores se emplean para inicializar los valores de los objetos y realizar las operaciones que sean necesarias para la generación de este objeto (crear otros objetos que puedan estar contenidos dentro de este objeto, abrir un archivo o una conexión de internet.....). Los constructores se comportan como cualquier otro método con estas dos peculiaridades: el nombre debe ser __init__ y no pueden devolver ningún tipo de dato.

Como cualquier método, un constructor admite sobrecarga. Cuando creamos un objeto (ya se verá más adelante como se hace) podemos invocar al constructor que más nos convenga.

```
class Animal:
  def __init__(self, age = "1", name = "dog"):
    self.age = age # public attribute
    self.name = name # public attribute
```

```
  def saluda(saludo='Hola', receptor = 'nuevo amigo'):
    print(saludo)



  def mostrarNombre():
    print(age)

  def mostrarEdad():
    print(name)
```

Es momento de pararnos y crear nuestra primera clase y construir nuestras dos primeras instancias (objetos) de dicha clase. Vamos a crear una clase coche, el constructor de la clase coche que va a recibir un único parámetro (la marca), y vamos a crear dos instancias coche cuyas marcas van a ser Mercedes y Audi respectivamente. Se hace notar que no se realiza la documentación, las excepciones ni las pruebas de los programas en estos ejemplos para facilitar su aprendizaje y no incrementar de forma excesiva la longitud del documento. Podemos imprimir los valores de dichos coches

```
class Car:
    def __init__(self, brand):
        self.brand = brand


def main():
    car1 = Car("Mercedes")
    car2 = Car("BMW")
    print("Coche 1:",car1.brand,"\ncoche 2:",car2.brand,
sep=" ")


if __name__ == "__main__":
    main()
```

1.2 Modificadores de métodos y variables

Antes de explicar herencia entre clases comentaremos cuales son los posibles modificadores que pueden tener métodos y variables y su comportamiento:

Modificadores de variables

- **public**: Pública, puede acceder todo el mundo a esa variable.
- **protected**: Protegida, sólo pueden acceder a ella las clases hijas de la clase que posee la variable y las que estén en el mismo package. Se definen mediante _ al inicio de su definición.
- **private**: Privada, nadie salvo la clase misma puede acceder a estas variables. Se definen mediante __ al inicio de su definición.
- **static**: Estática, esta variable es la misma para todas las instancias de una clase, **todas comparten ese dato**. **Si una instancia lo modifica todas comparten y ven dicha modificación**. También se les llama variables de clase. Estas variables se definen fuera de cualquier método de la clase. Estos métodos deben llamarse siempre con el nombre de la clase y el nombre del atributo. **Por ejemplo, para acceder a la variable estática de la clase animal, se haría mediante Animal.numAnimales**

```
class Animal:
  # variable de clase que se inicializa cuando se importa el
modulo animal
  numAnimales = 0
  def __init__(self, age = "1", name = "dog"):
    self.age = age # public attribute
    self.name = name # public attribute
    self.vivo = True
    Animal.numAnimales +=1

  def saluda(self, saludo='Hola', receptor = 'nuevo amigo'):
    print(saludo)



  def mostrarNombre(self):
    print(self.age)

  def mostrarEdad(self):
    print(self.name)

  def morir(self):
    if(self.vivo):
      self.vivo = False
```

```
        Animal.numAnimales -=1
```

Modificadores de un método

- **publ**ic: Público, puede acceder todo el mundo a este método.
- **protected**: Protegido, sólo pueden acceder a él las clases hijas de la clase que posea el método y las que estén en el mismo package. Se inicializa con un guion bajo _nombreMetodo
- **private**: Privada, nadie salvo la clase misma puede acceder a estos métodos. Se inicializa con dos guiones bajos __nombreMetodo
- **static**: Estática, es un método al cual se puede invocar sin crear ningún objeto de dicha clase. Math.sin, Math.cos son dos ejemplos de métodos estáticos. Desde un método estático sólo podemos invocar otros métodos que también sean estáticos. Se especifican con la etiqueta @staticmethod

A continuación mostramos un ejemplo de una clase que define métodos estáticos. Estos métodos se podrían invocar, por ejemplo, como Mat.cuadrado (4). En este caso, como son métodos estáticos (se instancian desde la clase), no hace falta que reciban el propio objeto que les llama ¡no hay self!).

Es decir, **podemos asumir que en Python todos aquellos atributos referentes a la instancia (diferentes objetos) deben llevar el self delante y deben estar definidas dentro de algún método, mientras que las variables de clase no llevan el self y están definidas en la clase fuera de cualquier método.**

En cuanto a los métodos, los métodos estáticos **deben llevar siempre la etiqueta @staticmethod**.

```
class Math:
    @staticmethod
    def cuadrado(i):
        return i*i
    @staticmethod
    def  mitad (i):
        return i/2
```

Otro ejemplo de un método estático con la clase Animal puede ser obtener el número de animales que tenemos. NumAnimales está definida fuera de cualquier función y no lleva el prefijo self. Age, name and vivo sí están definidas y accedidas en los diferentes métodos.

```
class Animal:
  # variable de clase que se inicializa cuando se importa el
modulo animal
  numAnimales = 0
  def __init__(self, age = "1", name = "dog"):
    self.age = age # public attribute
    self.name = name # public attribute
    self.vivo = True
    Animal.numAnimales +=1

  def saluda(self, saludo='Hola', receptor = 'nuevo amigo'):
    print(saludo + " " + receptor)

  def mostrarNombre(self):
    print(self.age)

  def mostrarEdad(self):
    print(self.name)

  def morir(self):
    if(self.vivo):
      self.vivo = False
      Animal.numAnimales -=1


  @staticmethod
  def get_numAnimales():
    return numAnimales
```

Volviendo al ejemplo del vector

En este apartado mostramos cómo se implementaría en Pyhon utilizando la POO una clase Vector que representa un vector en tres dimensiones y permita hacer sumas vectoriales y productos vectoriales. Se correspondería con la versión orientada a objetos del programa del cual había un ejercicio opcional en el Tema 1.

```
class Vector:

    def __init__(self, x, y=0, z=0):
        self._x = x
        self. _y = y
        self. _z = z

    def sumaVectorial(self, v):
        suma = Vector(self._x + v._x, self._y + v._y, self._z
+ v._z)
        return suma

    def multiplicacionVectorial(self, v):
        xm = self._y * v._z - self._z * v._y
        ym = -self._x * v._z + self._z * v._x
        zm = self._x * v._y - self._y * v._x
        multiplicacion = Vector(xm, ym, zm)
        return multiplicacion

    def __eq__(self, otherV):
        if(self._x == otherV._x and self._y == otherV._y and
self._z == otherV._z ):
            return True
        return False

    def __str__(self):
        return "(" + "x=" + self._x + ", y=" + self._y + ",
z=" + self._z + ")"
```

Este sería un ejemplo de código que usase esta clase:

```
def main():
    vector1 = Vector(1,1,1)
    vector2 = Vector(2,2,2)
    vector3 = Vector(3,3,3)
    vector4 = Vector(0,0,0)
```

```
    vectorResultAdd = vector1.sumaVectorial(vector2)
    vectorResultMult =
vector1.multiplicacionVectorial(vector2)
    if(vectorResultAdd == vector3):
        print("addVectors Test Passed")
    if(vectorResultMult == vector4):
        print("multVectors Test Passed")
```

En los ejemplos Vector2D y Vector se pueden ver explicaciones en los comentarios sobre cómo la herencia facilita la resolución del problema aplicando la herencia.

1.3 Herencia

Cuando en Python indicamos que una clase extiende (o hereda de) otra clase estamos indicando que es una clase hija de esta y que, por lo tanto, hereda todos sus métodos y variables. Este es un poderoso mecanismo para la reusabilidad del código. Podemos heredar de una clase, por lo cual partimos de su estructura de variables y métodos, y luego añadir lo que necesitemos o modificar lo que no se adapte a nuestros requerimientos. Veamos un ejemplo:

```
class Animal:
  # Class attribute. It is accessed by the name of the class
Animal.<class_attribute_name>
  # In our case, Animal.numAnimales. This attribute is seen by all
instances of Animal.
  numAnimales = 0

  def __init__(self, age = "0", name = "Godzilla"):
    '''constructor of an animal
    :param age: age of the animal: By default -> 0
    :param name: name of the animal. By Default: Godzilla
    :returns an instance of the Animal Class (One animal)
    '''
    # Todos estos atributos son públicos
    self.age = age # public attribute
    self.name = name # public attribute
    self.vivo = True
    Animal.numAnimales +=1

  # GETTERS AND SETTERS
  # Methods to access the attributes of the class
```

```
  def set_age(self, age):
    self.age = age

  def get_age(self):
    return self.age

  def set_name(self, name):
    self.name = name

  def get_name(self):
    return self.name

  def setVivo(self, vivo):
    self.vivo = vivo

  def isVivo(self):
    return self.vivo

#Methods of the Class. These methods are instantiated by INSTANCES
of the class
  def saluda(self, saludo='Hola', receptor = 'nuevo amigo'):
    print(saludo + " " + receptor)

# Take into account that all the class attributes are instantiated
by the own instance (self)
  def mostrarNombre(self):
    print(self.name)

  def mostrarEdad(self):
    print(self.age)

  def morir(self):
    if(self.vivo):
      self.vivo = False
      Animal.numAnimales -=1

#
  def __eq__(self, other):
    '''
    Overwrites the equality method of two instances of Animal.
    They are considered equal if they have the same name
```

```
    :param self, our own object where it is called the method from
    :param other, another object
    '''
    if other == None:
        return False
        if isinstance(other, Animal):
            if (self.get_name() == other.get_name()):
               return True
        return False

  def __str__(self):
    '''
    Overwrites the str method of the animal instance
    They are considered equal if they have the same name
    :param self, our own object where it is called the method from
    :param other, another object
    '''
    if(self.vivo == False):
        return "Soy " + self.name + " y tenia " + str(self.age) + "
years cuando fallecí"
    else:
        return "Soy " + self.name + " y tengo " + str(self.age) + "
years. ¡La vida pueda ser maravillosa!"

# An example of a static method. They are used to retrieve
information from the class, not from the instance.
  @staticmethod
  def get_numAnimales():
    return numAnimales
```

Herencia simple

Si un método no hace lo que nosotros queríamos podemos sobrescribirlo (overriding). Bastará para ello que definamos un método con el mismo nombre y argumentos. Para llamar a cualquier método de la clase Padre se puede:

- Utilizar la sintaxis super() y el nombre del método: **super().<NOMBRE_METODO>**
- Invocar al nombre de la clase y el nombre del método: **<NOMBRE_CLASE_PADRE>.<NOMBRE_METODO>**

Veámoslo sobre el ejemplo anterior:

```
from Animal import Animal
""" TODO DOC STRING OF THE MODULE IF IT CONTAINS MORE THAN ONE
CLASS (NOT RECOMMENDED)
"""

class Perro(Animal):
  """
  This class inheritates all attributes and methods from the Animal
Class
  A class used to represent a Dog
  We only specified in the docString the attributes and methods
defined in
  the class Perro, not in Animal!
  """
  # The only difference for the constructur is the default name,
but it may have different parameters or attributes
  def __init__(self, age = "0", name = "Toby", pastor = True):
    #Llamada al método de la clase padre
    # super().__init__(self, age, name)
    Animal.__init__(self, age, name)
    #Nuevos atributos si los tuviese
    super()
    self.pastor = pastor

#Nuevos getters y setters, solo para los atributos de la clase hija

  def set_pastor(self, pastor):
    self.pastor = pastor

  def is_pastor(self):
    return self.pastor

#Overwritten method
# Now, it says guau by default instead of Hola
  def saluda(self, saludo='Guau', receptor = 'nuevo amigo'):
    print(saludo + " " + receptor)

# An example of an inheritated static method
  @staticmethod
  def get_numAnimales():
```

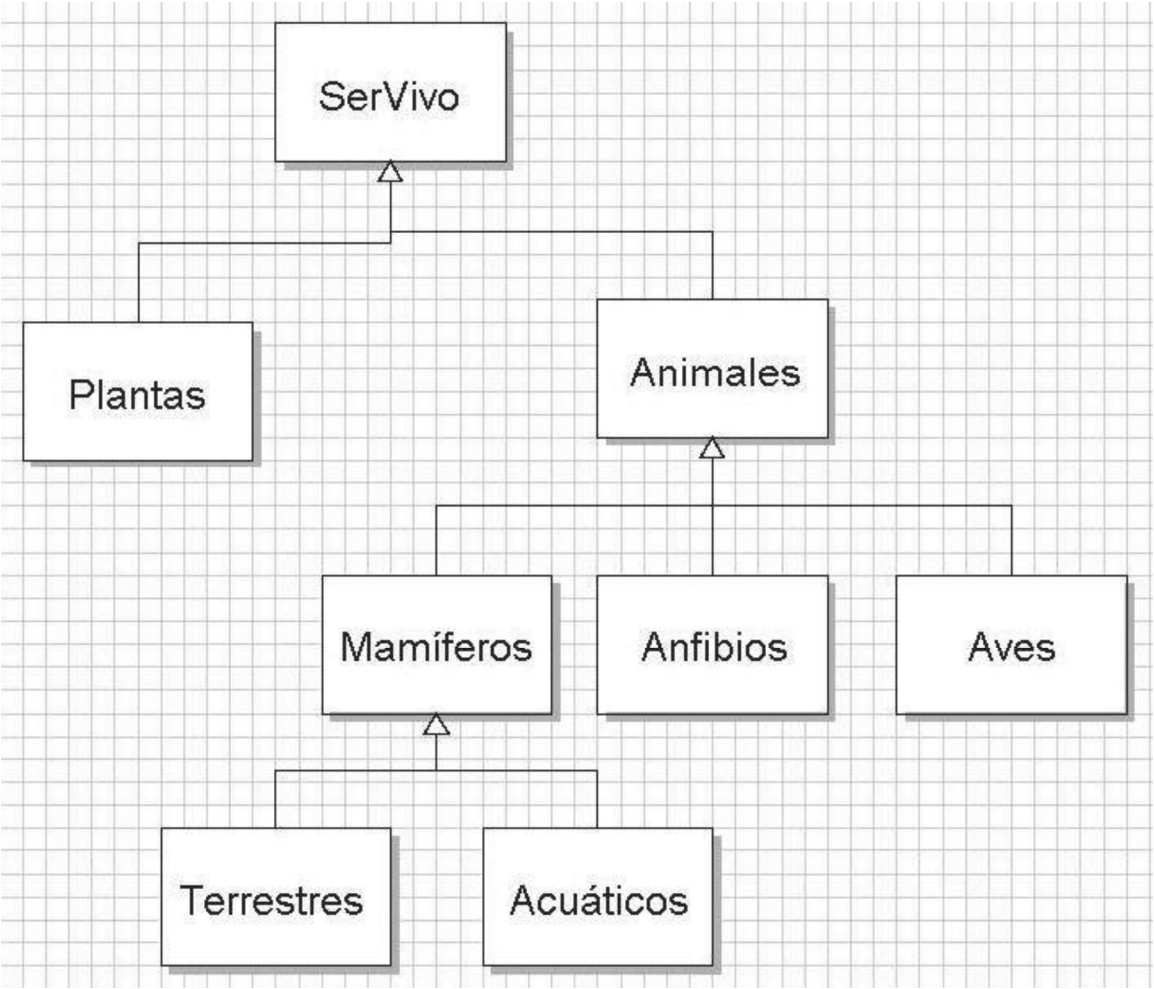
```
    print ("Este método no tiene sentido aquí, ya que está definido
en la clase padre")

def main():
# Aqui solo se definen las pruebas de la clase hija (perro), no de
la clase padre (Animal)
  perro = Perro("NuevoNombrePerro")
  #Como vemos, hereda el método __str__ de la clase Animal
  print(perro)
  perro.saluda(receptor="Clase de BI")
  perro.mostrarNombre()


if __name__=="__main__":
  main()
```

Aquí se puede ver como ahora el perro se imprime de manera diferente y saluda de manera diferente a un animal. Sin embargo, al heredar todas las propiedades de la clase Animal, puede realizar todas las acciones sin tener que implementarlas de nuevo.

Para sobreescribir un método que no hace lo que nosotros queríamos, podemos sobreescribirlo. Esto se hace mediante el polimorfismo de sustitución. Así, bastará para ello que definamos un método con el mismo nombre y los mismos argumentos que el método definido en la clase padre.

Jerarquía de herencias

Se pueden crear jerarquías completas (como un árbol genealógico) de clases. Como se puede ver en el tema 0 en el PDF de programación orientada a objetos, un perro tendrá todos los atributos de los mamíferos, y los mamíferos serán una especialización de los animales, que a su vez son una especialización de los seres vivos.



Por tanto, si queremos crear una clase Terrestre, esta debe heredar de Mamíferos, que a su vez ya ha heredado de Animales, y que a su vez ya ha heredado de SerVivo.

Herencia múltiple

A diferencia de lenguajes como Java y C#, el lenguaje Python permite la herencia múltiple, es decir, se puede heredar de múltiples clases. La herencia múltiple es la capacidad de una subclase de heredar de múltiples súper clases. Esto conlleva un problema, y es que, si varias súper clases tienen los mismos atributos o métodos, la subclase sólo podrá heredar de una de ellas. En estos casos Python dará prioridad a las clases más a la izquierda en el momento de la declaración de la subclase.

Esto puede ocurrir cuando una clase hereda de diferentes abstracciones sin relación entre sí.

Para llamar a los métodos o acceder a los atributos de la clase padre, se puede utilizar la misma sintaxis que en la herencia simple, pero en este caso hay que recordar la jerarquía que sigue Python de izquierda a derecha:

- Invocar a super() y el nombre del método: **super().<NOMBRE_METODO>. En este caso, Python buscará el método o el atributo de izquierda a derecha según la jerarquía de herencias fue establecida. En caso de no encontrarlo en la primera, buscará en la segunda, tercera, y así sucesivamente hasta que lo encuentre.**
- Invocar al nombre de la clase y el nombre del método: **<NOMBRE_CLASE_PADRE>.<NOMBRE_METODO>**

Por ejemplo, si representamos todo el reino animal, y dividimos los animales por nacionalidad además de por su ontología propia, un perro español debería heredar de terrestre, puesto que es un mamífero terrestre, y de la nacionalidad filipina, puesto que es de Filipinas y compartirá sus atributos.

En caso de que las clases Terrestre (o cualquier de sus clases Padre, que están contenidas en la clase hija) y Filipinas tuviesen atributos o métodos comunes, le especificamos la preferencia según el orden de izquierda a derecha. En este caso vamos a dar prioridad a la ontología de los animales respecto a la nacionalidad.

```
from Animal import Animal
from Nacionalidades import Filipinas
""" TODO DOC STRING OF THE MODULE IF IT CONTAINS MORE THAN ONE
CLASS (NOT RECOMMENDED)
"""

class Perro(Terrestre, Filipinas):
```

Aunque la herencia múltiple es una herramienta muy potente, no va a ser objeto de estudio durante el presente curso. Si algún estudiante está interesado en su utilización, puede sentirse libre de preguntar al profesor para más información.

1.4 Creación y referencia a objetos

Aunque ya hemos visto como se crea un objeto vamos a formalizarlo un poco. Un objeto en el ordenador es esencialmente un bloque de memoria con espacio para guardar las variables de dicho objeto. Crear el objeto es sinónimo de reservar espacio para sus variables, inicializarlo es dar un valor a estas variables. Para crear un objeto se utiliza el comando new. Veámoslo sobre un ejemplo:

```
class Fecha:
    ''' TODO CLASS DOCUMENTATION '''

    def __init__(self, day=1, month=1, year=1970):
```

```
        '''
        TODO DOCUMENTATION OF CONSTRUCTOR AND EXCEPTION
HANDLING (Type and value control)
        '''
        #DATETIME OF UNIX SISTEMS!
        self.__day=day
        self.__month = month
        self.__year = year

    def set_day(self, day):
        self.__day = day

    def get_day(self):
        return self.__day

    def set_month(self, month):
        self.__month = month

    def get_month(self):
        return self.__month

    def set_year(self, year):
        self.__year = year

    def get_year(self):
        return self.__year

    def __str__(self):
        return str(self.__day) + "/" + str(self.__month) +
str(self.__year)
```

Ahora podemos hacer:

```
From Fecha import Fecha
hoy = Fecha(29,10,2019)
print(hoy)
```

Con el primer comando hemos creado un puntero que apunta a una variable tipo Fecha, con los datos especificados en el constructor (29, 10, 2019)

Con esta sentencia creamos una variable que se llama hoy con valor 29-10-1970

Una vez creado un objeto será posible acceder a todas sus variables y métodos públicos, así por ejemplo en el ejemplo anterior hoy tomaría el valor 30. Si la variable fuese privada solo podrían acceder a ella sus instancias, por lo que hay que acceder a partir de sus métodos GET y SET:

```
hoy = Fecha(29,10,2019)
print(hoy)
if(str(hoy) == "29/10/2019"):
    print("Test Fecha.__str__ method passed")
else:
    print("Test Fecha.__str__ method failed")
hoy.set_day(30)
if(hoy.get_day() == 30):
    print("Test Fecha get and set day method passed")
else:
    print("Test Fecha get and set day method failed")
```

De este modo no podríamos acceder a las variables de la clase Fecha, para acceder a ella tendríamos que hacerlo mediante métodos que nos devolviesen su valor. A esto se le denomina encapsulación. Esta es la forma correcta de programar OOP: no debemos dejar acceder a las variables de los objetos por otro procedimiento que no sea paso de mensajes entre métodos, esto es, a través de los GETS y SETS si no se acceden desde dentro de la clase o desde una clase hija.

1.5 self

Es una variable especial de sólo lectura que proporciona Python. Contiene una referencia al objeto en el que se usa dicha variable. En Pyhton es imprescindible que un objeto se refiera a sí mismo como self.

Cuando necesitemos invocar a un método propio del objeto, debemos hacerlo con self. Cuando necesitemos invocar a un método estático lo haremos con el nombre de la clase Veamos dos ejemplos

```
class Customer:
    ''' TODO CLASS DOCUMENTATION '''
```

```
    def __init__(self, id, name):
        '''
        TODO DOCUMENTATION OF CONSTRUCTOR AND EXCEPTION
HANDLING (Type and value control)
        '''
        newAccount = self.createAccount(id)
        if(newAccount > 0):
            #DO WHATEVER TO CREATE THE CUSTOMER
            print("Welcome, Mr. " + name)
        else:
            print("Sorry, but you are not elegible to be a
customer right now. Call to XXXXXX for personal assistance")

    def createAccount(self, id):
        '''
        TODO DOCUMENTATION OF CONSTRUCTOR AND EXCEPTION
HANDLING (Type and value control)
        '''
        #If everything was correct return the new
accountNumber, if not, return 0
        accountNumber=100 # All the logic implementation is
abstracted
        wasCorrect=True
        if(wasCorrect):
            return accountNumber
        else:
            return 0
```

Otro posible uso de this, que ya se ha visto en ejemplos anteriores es diferenciar entre variables locales o parámetros de un método o constructor y variables del objeto. En los códigos correspondientes a los ejemplos Animal.py y Perro.py el constructor de la clase Animal o Perro aunque realiza la misma función que los que se recogen en estos puntos, son ligeramente diferentes. Las variables que se llaman con self son propias del objeto desde el que se invoca, mientras las variables sin self son variables locales o parámetros.

```
  def __init__(self, age = "0", name = "Godzilla"):
    self.age = age # public attribute
    self.name = name # public attribute
```

```
    self.vivo = True
```

1.6 Constructor de la clase padre

Siempre que se hereda de una clase, es necesario llamar al constructor (__init__) de la clase padre para inicializar los atributos allí definidos.

```
  def __init__(self, age = "0", name = "Toby", pastor =
True):
    #Llamada al método de la clase padre
    Animal.__init__(self, age, name)
    #Nuevos atributos si los tuviese
    self.pastor = pastor
```

1.7 Un ejemplo de herencia con la clase Vector

En este apartado vamos a ver cómo podemos modificar la clase Vector para, apoyándonos en ella, representaron vector en un espacio de dos dimensiones. La suma de vectores en dos dimensiones debe ser un vector en dos dimensiones. Pero el producto vectorial de vectores en dos dimensiones debería ser un vector perpendicular, y por tanto en tres dimensiones. Y también sería deseable poder sumar vectores en dos dimensiones con vectores en tres dimensiones, obteniendo obviamente un vector de tres dimensiones.

```
from Vector import Vector

# Vector2D es un Vector en 3D donde la z siempre es 0
class Vector2D(Vector):

    def Vector2D(self, x, y):
        '''
        TODO EXCEPTION HANDLING, TYPE CHECKING, DOCUMENTATION
        '''
        Vector.__init__(self, x , y, 0)
    #Sobreescribimos el método de suma
```

```
    def sumaVectorial(self, otherVector):
        #The attributes _x and _y are protected in the class
Vector!
        suma = Vector2D(self._x + otherVector._x, self._y +
otherVector._y);
        return suma
    #Sobre escribimos la representación textual
    def __str__(self):
        return "(" + "x=" + str(self._x) + ", y=" +
str(self._y) + ")"
```

Observa como hemos tenido que hacer menos trabajo al apoyarnos en la clase Vector. A continuación, mostramos un código que opera con vectores 2D, que también combina operaciones con vectores en 2 y 3 dimensiones:

```
def main():
    vector1 = Vector2D(1,0)
    vector2 = Vector2D(0,1)
    vector3 = Vector(0,0,1)
    vectorResultAdd = vector1.sumaVectorial(vector2)
    print(vectorResultAdd)
#Un Vector2D es un Vector así que puede multiplicarse con ellos
#en esta ocasion el metodo que se invoca es el del padre
    vectorResultMult =
vectorResultAdd.multiplicacionVectorial(vector3)
    print(vectorResultMult)
#Y tambien podemos sumar un Vector2D con un Vector 3D
#en esta ocasion el metodo que se invoca es el del padre
    vectorAddV3V2 =
vectorResultMult.sumaVectorial(vectorResultAdd);
    print(vectorAddV3V2)
```

1.8 Enumerados (Enum)

Un enumerado (o Enum) es una clase "especial" (tanto en Python como en otros lenguajes) que limitan la creación de objetos a los especificados explícitamente en la implementación de la clase.

Esto sirve para restringir los posibles valores de entrada de un atributo perteneciente a una clase. Un ejemplo es los tipos de guerreros que existen en nuestra Guerra entre terrícolas y marcianos. Para limitar el tipo de guerreros a marcianos y terrícolas, se debe crear una enumeración con estos dos valores, que van a ser los únicos permitidos en la clase guerreros.

Los miembros (MARCIANO, TERRICOLA) de las enumeraciones (Guerreros) no son enteros ni cadenas, sino que son objetos con dos atributos: nombre y valor. El nombre es el primer elemento (MARCIANO, TERRICOLA) y el valor el segundo elemento (1,2, respectivamente).

Las enumeraciones son iterables y son accesibles a partir de la clave igual que un diccionario en la tupla nombre, valor.

Las enumeraciones no pueden tener dos miembros con el mismo nombre, pero sí pueden tener miembros con el mismo valor.

La comparación entre elementos de un enumerado se hace a partir de su clave. Si dos elementos tienen la misma clave, estas van a ser consideradas iguales con el comparador ==, aunque no lo serán con el método isInstance

```
import enum
# Using enum class create enumerations
class Guerreros(enum.Enum):
   MARCIANO = 1
   TERRICOLA = 2

class GuerrerosTypeError(Exception):
   """Raised when the guerreros type is wrong"""
   pass
```

2 Nuestro primer programa orientado a objetos

Para intentar hacer este tema un poco menos teórico y ver algo de lo que aquí se ha expuesto se ha realizado el siguiente programilla; en él se empieza una ficticia guerra entre dos naves, una de marcianos y otra de terrícolas, cada uno de los cuales va disparando, generando números aleatorios, y si acierta con el número asignado a algún componente de la otra nave lo "mata". No pretende ser en absoluto ninguna maravilla de programa, simplemente se trata con él de romper el esquema de programación estructurada o clásica, en el cual el programa se realiza en el main llamando a funciones. En OOP un programa, digámoslo una vez más, es un conjunto de objetos que dialogan entre ellos pasándose mensajes para resolver un problema. Veremos cómo, ni más ni menos, esto es lo que aquí se hace para resolver un pequeño problema-ejemplo.

Antes de pasar al código, veamos la representación UML del diagrama de clases que componen nuestro problema.

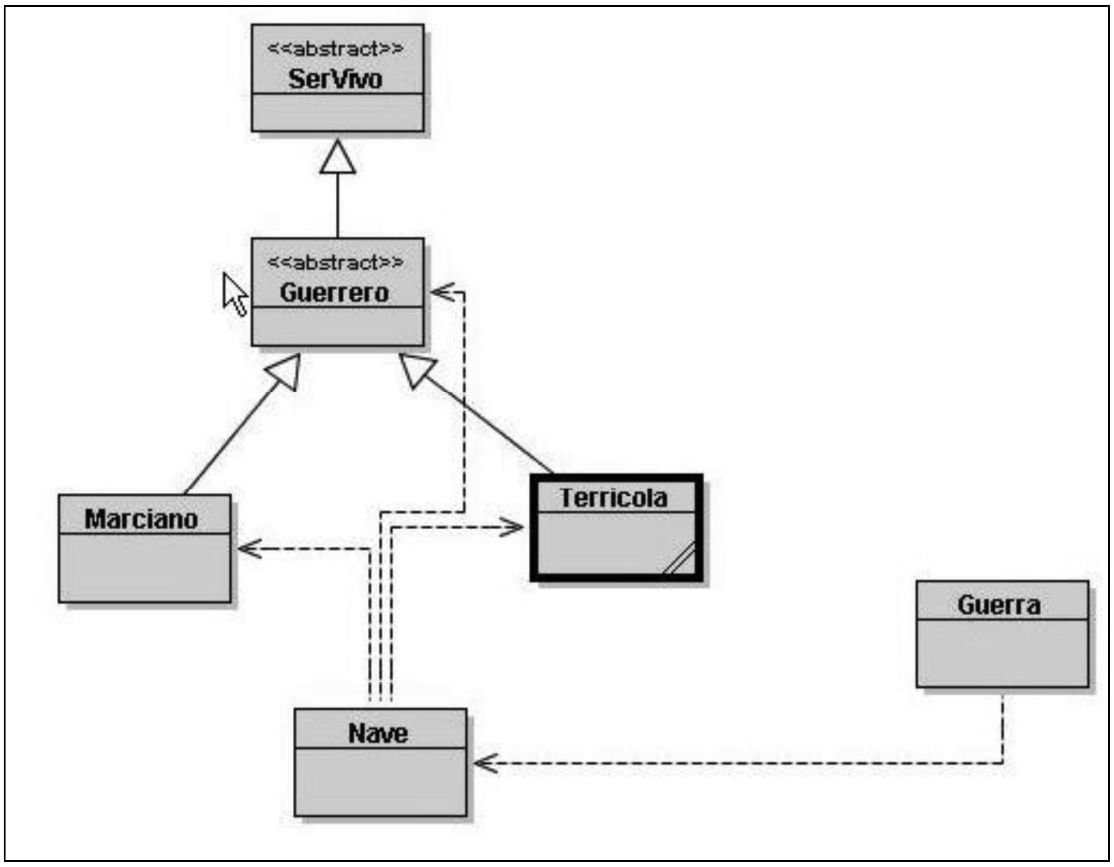


Figura 1.- Diagrama de clases de nuestro problema visualizado. Contamos con una clase SerVivo, que representa a un ser vivo. Guerrero es nuevamente una clase que hereda de SerVivo (un guerrero es un ser vivo); en ella se sitúa el comportamiento de atacar. Tenemos dos clases, Terrícola y Marciano, que representarán a terrícolas y marcianos. La clase Nave actúa como un contenedor cuya tripulación puede ser un conjunto de terrícolas o de marcianos. Por último, Guerra es una clase auxiliar encargada de la creación de las naves y de la simulación de la guerra.

La clase SerVivo simplemente representa a un ser que está vivo:

```
class SerVivo:

    def __init__(self):
        self._vivo = True

    def is_vivo(self):
        return self._vivo

    def morir(self):
        self._vivo= False
```

La clase Guerrero representa un ser vivo con capacidad de combatir; tiene una variable donde se guarda un número aleatorio entre 0 y 9; si alguien acierta con ese número el guerrero se morirá. Además, tiene capacidad para disparar, esto es, generar números aleatorios que representan un disparo a un adversario (método shoot()). También tiene un método que se invoca cuando alguien le ha

disparado a él y comprueba si tiene o no que morirse (si han acertado o no con su número aleatorio). Observa como también hay un método privado; este método privado forma parte de un detalle de implementación de la clase y ninguna otra clase necesita conocerlo.

```
from SerVivo import SerVivo
from Utilidades import *

class Guerrero(SerVivo):
    __max_target = 10

    def __init__(self, name):
        SerVivo.__init__(self)

        self._target = self.__generateTargetToDie()
        self._name = name

    def get_name(self):
        return self._name
    # No se crea un set porque los guerreros no pueden cambiar de nombre
    '''
    def set_name(self, name):
        return self._name = name
    '''

    def get_target(self):
        return self._target

    def shoot(self):
        '''
        Shoot if the warrior is alive generating a randome number between 0
and the ____max_target
        :returns the number to shoot if the warrior is alive, -1 otherwise
        '''
        if(self._vivo):
            shot = generaIntAleatorio(0,Guerrero.__max_target)
            print(self._name + " dispara " + str(shot))
            return shot
        else:
            return -1

    def get_shot(self, shot):
        '''
        If the target is guessed by the shoot, then the warrior dies.
        :param shoot: int with the shoot against the soldier
```

```
        :returns True if the shot kills the warrior (shot is the target and
the warrior is alive), False otherwise
        '''
        isTarget = False
        if(self._vivo == True and self._target == shot):
            self._vivo = False # The SerVivoDIES!
            isTarget = True
            print(self._name + " se muere por el disparo " + str(shot))
        return isTarget

    def __generateTargetToDie(self):
        '''
        Private method to generate the target to get shot
        '''
        return generaIntAleatorio(0,Guerrero.__max_target)

    ''' TODO IF NEEDED OVERRIDE METHODS EQUALITY, COMPARISON, HASH , etc.
    '''
    def __str__(self):
        '''Override method toString to identify the objects and know their
states
        '''
        return self._name

    @staticmethod
    def get_maxTarget():
        return Guerrero.__max_target
```

La clase Marciano representa a los marcianos. **Los marcianos emplean una variable de clase (__totalMarcianosAlive) para contar cuántos marcianos** quedan vivos en cada momento. Esta variable de clase se **incrementa en su constructor**. Su constructor también llama al constructor de Guerrero y le pasa la cadena de caracteres que contiene el texto informativo de qué tipo de Guerrero se trata (un marciano en este caso). Observa como además **Marciano modifica la forma en la que los guerreros reciben los disparos: los marcianos son más duros de matar y hay que dispararles un número de veces igual a __shotsToKillAMarciano** para conseguir matarlo. Una vez que se le ha disparado ese número de veces, el método get_shot () del Marciano llama al método get_shot () de Guerrero, que es el que realmente se encarga de ejecutar el código necesario para "matar" al Marciano y mostrar el mensaje correspondiente en pantalla.

```
class Marciano(Guerrero):
    __totalMarcianosAlive = 0
    __shotsToKillAMarciano = 3
```

```
    def __init__(self, name):
        Guerrero.__init__(self, name)
        self.__shotsToReceive = Marciano.__shotsToKillAMarciano
        Marciano.__totalMarcianosAlive += 1

    def get_shotsToReceive(self):
        return self.__shotsToReceive
    # Overrides the method get_shot from the parent class!
    def get_shot(self, shot):
        '''
        If the target is guessed by the shoot, then the warrior
dies.
        :param shoot: int with the shoot against the soldier
        :returns True if the shot kills the warrior (shot is the
target and the warrior is alive), False otherwise
        '''
        isTarget = False
        if(self._vivo == True and self._target == shot):
            self.__shotsToReceive -=1
            isTarget = True
            if(self.__shotsToReceive == 0):
                Guerrero.get_shot(self,shot)
                Marciano.__totalMarcianosAlive -= 1

        return isTarget

    @staticmethod
    def get_total_marcianos_alive():
        '''
        :returns the total number of Marcianos alive
        '''
        return Marciano.__totalMarcianosAlive

    @staticmethod
    def get_total_marcianos_alive():
        '''
        :returns the shots needed to kill a marciano
        '''
        return Marciano.__shotsToKillAMarciano
```

La clase Terrícola también tiene una variable estática que se emplea para contar cuántos terrícolas quedan vivos en cada momento (__totalTerricolasAlive). Observa como se ha sobrescrito el método get_shot (); en este caso no estamos realmente cambiando el comportamiento del método de la clase padre (Guerrero); simplemente estamos haciendo otra cosa adicional: decrementar el número total de terrícolas que quedan vivos. Pero los terrícolas se mueren igual que los guerreros. Sin embargo, los terrícolas no disparan igual que los guerreros. Los terrícolas son más inteligentes a la hora de disparar, y llevan cuenta de los "disparos" realizados; esto es, de los números aleatorios que han generado. Cuando generan un número aleatorio un número de veces igual a Marciano. __shotsToKillAMarciano entonces no vuelven a generar nunca más ese número aleatorio; generar de nuevo ese número aleatorio es inútil porque todos los marcianos que se pueden "matar" con ese número ya se han muerto. Por tanto, si ese número aleatorio se vuelve a generar "desechan el disparo y disparan de nuevo".

Es interesante observar en las clases Terrícola y Marciano como ambos pueden "disparar" y "recibir disparos", y ambas lo hacen a través de los mismos métodos (que están definida en la clase Guerrero), pero ambas lo hacen de modo diferente. Los terrícolas disparan de un modo más inteligente, pero se mueren con un solo disparo. Los marcianos disparan de un modo más tonto (pueden repetir números aleatorios) pero son más resistentes y requieren tres disparos para morirse.

```
class Terricola(Guerrero):

    __totalTerricolasAlive = 0
    __shots_done = [0] * (Guerrero.get_maxTarget() + 1)

    def __init__(self, name):
        Guerrero.__init__(self, name)
        Terricola.__totalTerricolasAlive += 1

    # Overrides the method shoot from the parent class!
    def shoot(self):
        '''
        Shoot if the Terricola is alive generating a random number between
0 and the __max_target
        It shoots a number of times equals to the number of shoots that a
Marciano can hold
        :returns the number to shoot if the warrior is alive, -1 otherwise
        '''
        if(self._vivo):
            shot = generaIntAleatorio(0,Guerrero.get_maxTarget())
            while(Terricola.__shots_done[shot] >=
Marciano.get_shotsToKillAMarciano()):
                shot = generaIntAleatorio(0,Guerrero.get_maxTarget())
            Terricola.__shots_done[shot] += 1
```

```
            print(self._name + " shoot " + str(shot) + " for " +
str(Terricola.__shots_done[shot]) + " time")

            return shot
        else:
            return -1

    # Overrides the method get_shot from the parent class!
    def get_shot(self, shot):
        '''
        If the target is guessed by the shoot, then the warrior dies.
        :param shoot: int with the shoot against the soldier
        :returns True if the shot kills the warrior (shot is the target and
the warrior is alive), False otherwise
        '''
        isTarget = Guerrero.get_shot(self,shot)
        if(self._vivo):
            __totalTerricolasAlive -= 1

        return isTarget

    @staticmethod
    def get_total_Terricolas_alive():
        '''
        :returns the total number of Terricolas alive
        '''
        return Terricola.__totalTerricolasAlive

    @staticmethod
    def get_total_shots_done(shoot):
        '''
        :returns the total number of Terricolas alive
        '''
        if(not isinstance(shoot,int)):
            raise TypeError("shoot should be an int")
        elif(shoot < 0 or shoot > Guerrero.get_maxTarget()):
            raise ValueError("shoot OUT OF RANGE")

        return Terricola.__shots_done[shoot]
```

La clase Nave hace de contenedor y contiene una tripulación de guerreros, que pueden ser o bien marcianos o bien terrícolas. Observa cómo el constructor de la clase nave, en función de la cadena de caracteres que se le pasa como argumento, decide si tiene que crear Marcianos o Terrícolas para

guardar en el array de Guerreros. Las naves también son las que generan los disparos; para ello le piden a un tripulante que genere el disparo (generaDisparo(int tripulante)). También son ellas las que reciben inicialmente el disparo (recibeDisparo(int disparo)), y notifican a todos los tripulantes de que han recibido un disparo para que éstos decidan si tienen o no que hacer algo (morirse si han adivinado su número aleatorio y se trata de un terrícola, o decrementar su contador de disparos recibidos si se trata de un marciano). Observa cómo el objeto Nave fundamentalmente delega en su tripulación para todas sus actividades; es la tripulación la que dispara, y es la tripulación la que recibe los disparos.

```
class Nave():

    def __init__(self, warrior_type, name="Millenium Falcon", crew=1):
        if(not isinstance(crew,int)):
            raise TypeError("crew is the number of warriors and should be
an int")
        elif(crew < 0):
            raise ValueError("crew is the number of warriors and should be
an int > 0")
        elif(warrior_type == Guerreros.MARCIANO):
            self.__type = warrior_type
            self.__name = name
            self.__crew = list()
            for i in range (0,crew):
                marciano_name="marciano" + str(i)
                self.__crew.append(Marciano(marciano_name))
        elif(warrior_type == Guerreros.TERRICOLA):
            self.__type = warrior_type
            self.__name = name
            self.__crew = list()
            for i in range (0,crew):
                terricola_name="terricola" + str(i)
                self.__crew.append(Terricola(terricola_name))
        else:
            raise GuerrerosTypeError("TYPE of GUERREROS WRONG")

        print("Created a ship " + self.__name + " of " + str(self.__type) +
" with " + str(crew) + " members")

    def __str__(self):
        return self.__name + " OF " + str(len(self.__crew)) + " " +
str(self.__type)

    def get_shot(self,shot):
        if(not isinstance(shot,int)):
```

```
            raise TypeError("shot should be an int")
        elif(shot == -1):
            # A shot from a dead member
            pass
        elif(shot < 0 or shot > Guerrero.get_maxTarget()):
            raise ValueError("shot OUT OF RANGE")
        for i in range(0,len(self.__crew)):
            if(self.__type == Guerreros.MARCIANO or self.__type ==
Guerreros.TERRICOLA):
                self.__crew[i].get_shot(shot)
            else:
                raise GuerrerosTypeError("TYPE of GUERREROS WRONG in the
crew " + self.__crew)

    def shoot(self,warrior):

        if(warrior >= 0 and warrior < len(self.__crew)):
            return self.__crew[warrior].shoot()
        else:
            raise ValueError("The warrior: " + str(warrior) +  " does not
exist in the ship: " + self.__name)

    def membersAlive(self):

        membersAlive=0
        # I AM LAUNCHING AN IMPOSSIBLE EXCEPTION, SINCE IT CANT BE ANY NAVE
WITH OTHER TYPE THAN MARCIANO OR TERRICOLA
        # Read Constructor to double check it. That is why the exception is
not documented
        if(self.__type == Guerreros.MARCIANO or self.__type ==
Guerreros.TERRICOLA):
            for warrior in self.__crew:
                #We access the member of the grandparent class (SerVivo),
that is shared by all warriors (Marciano and Terricola)!
                if(warrior.is_vivo()):
                    membersAlive+=1
        else:
            raise GuerrerosTypeError("TYPE of GUERREROS WRONG ")

        return membersAlive

    def isWarriorAlive(self, warrior):
        if(warrior >= 0 and warrior < len(self.__crew)):
            return self.__crew[warrior].is_vivo()
        else:
```

```
            raise ValueError("The warrior: " + str(warrior) +  " does not
exist in the ship: " + self.__name)

    def number_of_members(self):
        return len(self.__crew)
```

La última clase es Guerra; esta será la clase que cree las dos naves que van a combatir. Y es la que hace que las dos naves "se peleen". En esta guerra sólo hay dos naves, y la una dispara a la otra. Pero sería trivial extender este código para tener una flota de naves de terrícolas y de marcianos que participasen en una misma guerra.

```
class Guerra():

    def __init__(self, numTerricolas=3, numMarcianos=3):
        self.__terricolasNave = Nave(Guerreros.TERRICOLA, "Millenium
Falcon", numTerricolas)
        self.__marcianosNave = Nave(Guerreros.MARCIANO, "Wing X",
numTerricolas)

    def start_war(self):
        numTerricolasInNave = self.__terricolasNave.number_of_members()
        numMarcianosInNave = self.__marcianosNave.number_of_members()
        if(numTerricolasInNave >= numMarcianosInNave):
            max_shots = numTerricolasInNave
        else:
            max_shots = numMarcianosInNave
        while(self.are_members_in_both_crews()):
            for warrior in range (0,max_shots):
                if(warrior < numTerricolasInNave and
self.__terricolasNave.isWarriorAlive(warrior)):
                    shot = self.__terricolasNave.shoot(warrior)
                    self.__marcianosNave.get_shot(shot)
                if(warrior < numMarcianosInNave and
self.__marcianosNave.isWarriorAlive(warrior)):
                    shot = self.__marcianosNave.shoot(warrior)
                    self.__terricolasNave.get_shot(shot)
        if (self.__terricolasNave.membersAlive() > 0):
            return 0
        elif (self.__marcianosNave.membersAlive() > 0):
            return 1
        else:
            raise Exception
```

```
    def are_members_in_both_crews(self):
        if( (self.__terricolasNave.membersAlive()) > 0 and
(self.__marcianosNave.membersAlive()) > 0):
            return True
        else:
            return False
```

3 Aprendiendo a usar los módulos y los paquetes

A estas alturas deberías tener claro que una clase tiene una **parte privada que oculta a los demás y que no es necesario conocer para poder acceder a la funcionalidad de la clase**. **Si hacemos cambios a la parte privada de la clase, mientras se respete la parte pública, cualquier código cliente que emplee la clase no se dará cuenta de dichos cambios**.

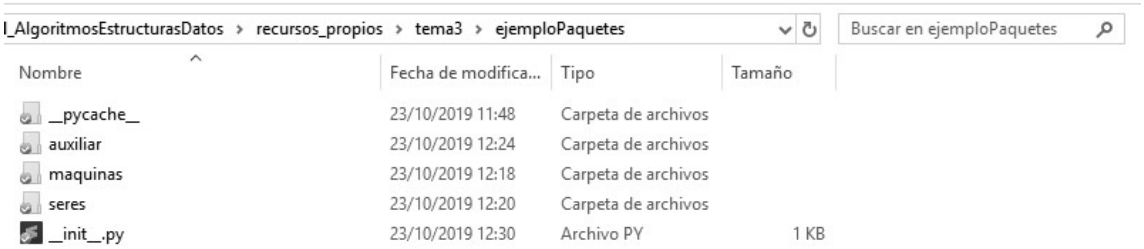
Imagínate que tú y un compañero vais a construir en un programa complejo juntos. Os repartís el trabajo entre los dos y cada uno de vosotros implementa su parte como un montón de clases Python. Cada uno de vosotros en su código va a emplear parte de las clases del otro. Por tanto, os ponéis de acuerdo en las interfaces (métodos públicos) de esas clases. Sin embargo, cada uno de vosotros para construir la funcionalidad de esas clases probablemente se apoye en otras clases auxiliares. A tu compañero le dan igual las clases auxiliares que tú emplees. Es más, dado que el único propósito de esas clases es servir de ayuda para las que realmente constituyen la interfaz de tu parte del trabajo sería contraproducente que él pudiese acceder a esas clases que son detalles de implementación: tú en el futuro puedes decidir cambiar esos detalles de implementación y cambiar esas clases, modificándolas o incluso eliminándolas.

Dada esta situación ¿no sería interesante poder "empaquetar" tu conjunto de clases de tal modo que ese "paquete" sólo dejase acceder a tu compañero a las clases que tú quieras y oculte las demás? Esas clases a las que se podría acceder serían la interface de ese "paquete". Serían "públicas". Dentro del paquete tú puedes meter cuantas más clases quieras. Pero esas no serán vistas por tu compañero y podrás cambiarlas en cualquier momento sin que él tenga que modificar su código. Es la misma idea que hay detrás de una clase pero llevada a un nivel superior: una clase puede definir cuáles de sus partes son accesibles y no accesibles para los demás. El paquete permitiría meter dentro cuantas clases quisieras pero mostraría al exterior sólo aquellas que considere adecuado.

Esta es precisamente la utilidad de los *package* en Python. Empaquetar un montón de clases y decidir cuáles serán accesibles para los demás y cuáles no. Para empaquetar las clases simplemente debemos poner al principio del archivo donde definimos la clase, en las dos primeras líneas que no sean un comentario, una sentencia que incluya en el path los paquetes que vamos a importar, y que incluye todos los subpaquetes. A continuación, debemos indicar los subpaquetes y módulos indique a qué paquete pertenece:

```
import sys
sys.path.insert(0, '<PATH_OF_MY_PACKAGE>')
```

Una clase que esté en el paquete "auxiliar" debe situarse dentro de un directorio con nombre "auxiliar". En Python los paquetes se corresponden con una jerarquía de directorios. Por tanto, si para construir un programa quiero emplear dos paquetes diferentes con nombres "auxiliar" y "maquinas" en el directorio de trabajo debo crear dos subdirectorios con dichos nombres y colocar dentro de cada uno de ellos las clases correspondientes. En la figura, el directorio de trabajo desde el cual deberíamos compilar y ejecutar la aplicación es "paquetes". En cada uno de los dos subdirectorios colocaremos las clases del paquete correspondiente.



Cuando una clase se encuentra dentro de un módulo que a su vez está dentro de un paquete el nombre de la clase pasa a ser "nombrepaquete.NombreModulo.NombreClase". Así, la clase "Guerra" que se encuentra físicamente en el directorio " auxiliar" se debe importar con la siguiente instrucción.

```
import sys
sys.path.insert(0, '<PATH_OF_MY_PACKAGE>/<Package_name>/')

from subpackageName.ModuleName import Class #General
from auxiliar.Guerra import Guerra #Ejemplo
```

Para todos los efectos, **el nombre de la clase es " paquete.subpaquete.modulo.clase".** Cuando en una clase **no se indica que está en ningún paquete**, como hemos hecho hasta ahora en todos los ejemplos de este tutorial, esa clase se sitúa en el **directorio built-in de Python, o en el directorio donde se ejecuta el programa (default package**). En ese caso, el nombre de la clase es simplemente lo que hemos indicado después de la palabra reservada class sin precederlo del nombre de ningún paquete.

Es posible anidar paquetes; por ejemplo, en el directorio "paquete1" puedo crear otro directorio con nombre "paquete11" y colocar dentro de él la clase "OtraClase". La primera línea de dicha clase debería ser:

```
package paquete1.paquete11;
```

y el nombre de la clase será "paquete1.paquete11.OtraClase".

¿Cómo indico qué clases serán visibles en un paquete y qué clases no serán visibles?. Cuando explicamos cómo definir clases, atributos y métodos vimos que antes de la definición se puede especificar con un guión bajo o dos guiones bajos si es private, protected o public. Es decir, podíamos poner un modificador de visibilidad. Hasta ahora siempre hemos empleado el modificador public (sin guiones) para las clases. Ese modificador significaría que la clase va a ser visible desde el exterior, forma parte de la interfaz del paquete.

Por tanto, poniendo o no poniendo los modificadores _ y __ podemos decidir qué forma parte de la interfaz de nuestros paquetes y qué no.

¿Y cómo hacemos para emplear clases que se encuentren en otros paquetes diferentes al paquete en el cual se encuentra nuestra clase? Para eso es precisamente para lo que vale la sentencia import. Para indicar que vamos a emplear clases de paquetes diferentes al nuestro. Así, si desde la clase "MiClase" que se encuentre definida dentro de "paquete1" quiero emplear la clase "OtraClase" que se encuentra en "paquete2" en "MiClase" debo añadir la sentencia:

```
from paquete import clase
from paquete.subpaquete import clase
from paquete.subpaquete import clase2.metodo
```

A partir de ese momento, **si clase era pública, podré acceder a ella y crear instancias.** El importar una clase sólo será posible si dicha clase forma parte de la interfaz pública del paquete. También podemos escribir la sentencia:

```
from paquete import *
```

Que haría accesibles todos los módulos y sus clases públicas que se encuentren en "paquete", y no sólo una como el ejemplo anterior,. En este caso se importarían todos los subpaquetes y módulos contenidos en el paquete paquete.

3.1 Un ejemplo de código con paquetes

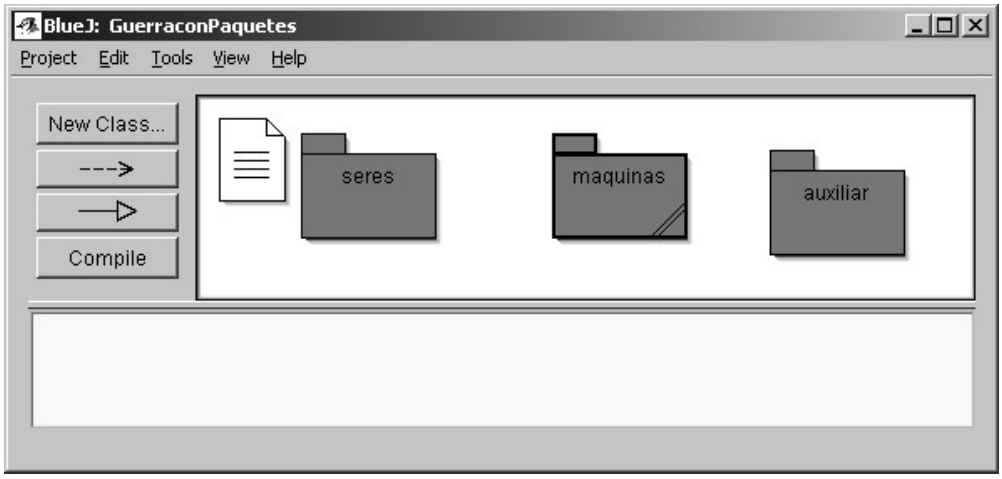
Vamos a ver un código en el que se ponen un uso los conceptos que estamos presentando aquí. Este código se encuentra en el directorio "ejemploPaquetes".

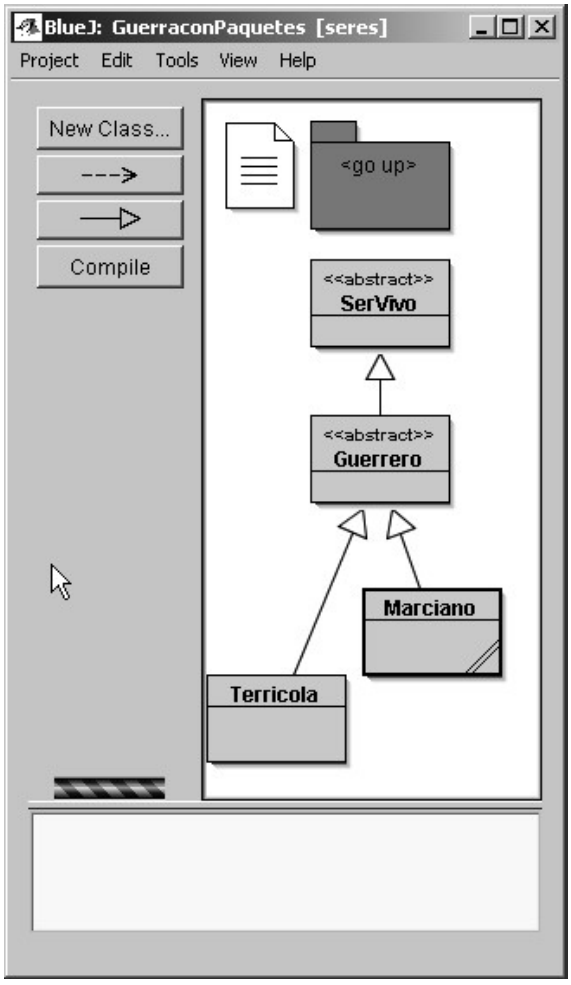
3.2 El ejemplo de los marcianos con paquetes

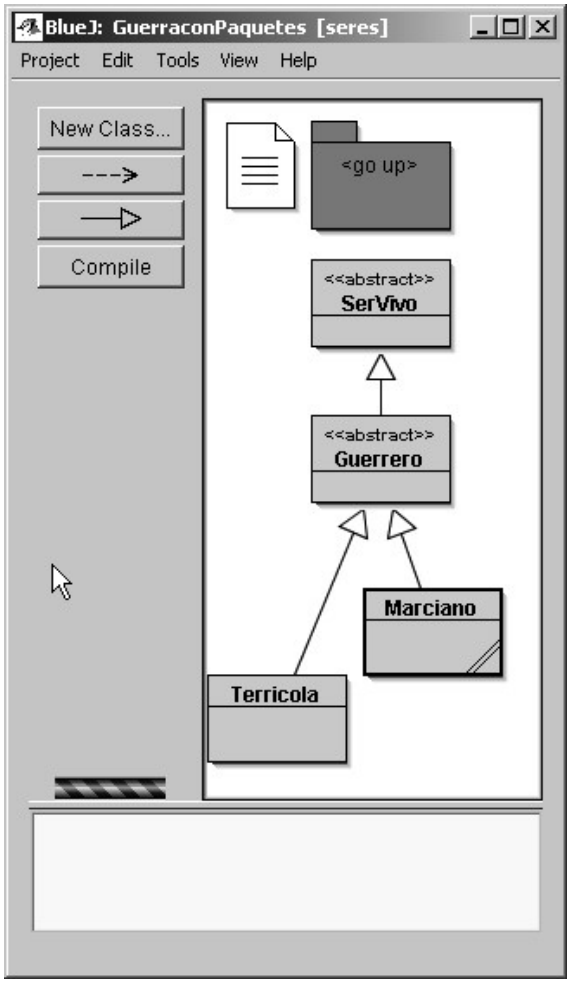
En esta sección vamos a reorganizar el código de la guerra entre marcianos y terrícolas para emplear paquetes. La principal funcionalidad de los paquetes es precisamente ayudar a organizar programas complejos. El cómo organizar un código en paquetes depende completamente del propósito de dicho código y de cómo se quiera diseñar el software. Con el tiempo irás ganando experiencia e irás aprendiendo a organizar tus clases en paquetes de modo adecuado.

Para un ejemplo tan trivial como éste podría perfectamente argumentarse que con emplear un único paquete es más que suficiente. No obstante, vamos a reorganizar el código empleando tres paquetes. Uno de los paquetes lo llamaremos "seres" porque en él será donde vayan todos los objetos del universo que consideramos "seres". En nuestro caso, irán Marciano, Terrícola, Guerrero y SerVivo. Crearemos un segundo paquete llamado "máquinas" donde irían todas las máquinas que implementásemos. En nuestro caso, la única máquina en el problema es Nave. Por último, creamos un paquete "auxiliar" en el cual colocamos la clase que contiene el método main.

Los únicos cambios que será necesario hacer en el código es incluir las sentencias que indican a qué paquete pertenece cada clase, colocar cada clase en el directorio adecuado y, cuando una clase emplee clases de otros paquetes diferentes al suyo, añadir la sentencia import correspondiente. Puedes consultar el código fuente en los códigos de ejemplo que vienen con los apuntes; aquí vamos a admitirlo porque es prácticamente idéntico al ejemplo ya presentado. La apariencia de los tres paquetes que forman el programa será:







4 Excepciones

A estas alturas te habrás dado cuenta que gestionar los posibles errores que pueden suceder en un programa es una de las tareas más tediosas y complicadas que debe realizar un programador. En C la única forma que tenemos de comunicar que ha sucedido un error en un módulo o una función es devolver un código de error; esto es, hacer que la función en vez de devolver un dato válido devuelva un

dato imposible que será interpretado como un error por el código que la invoca. Por ejemplo, si tenemos una función que debe devolver el tamaño de un array dinámico y esta función se invoca antes de que el array haya sido inicializado la función podría devolver "-1".

Este mecanismo de gestión de errores es extremadamente débil y limitado. ¿Qué pasaría si la función pudiese devolver cualquier valor? Por ejemplo, si una función devuelve la suma de dos números ¿Cómo podría la función indicarle al código que le invoca que se ha producido un overflow? Por otro lado, ahora que estamos programando en Python ¿qué pasa si algo va mal dentro de un constructor? Los constructores, por definición, nunca devuelven nada. No es posible chequear el valor devuelto. Además, el hecho de que devolvamos un código de error no garantiza que quien ha invocado al método correspondiente le preste atención. Por último, si sólo contamos con este mecanismo de gestión de errores cuando nos enteramos de que ha sucedido un error a veces es demasiado tarde para hacer nada. Por ejemplo, imaginemos que intentamos hacer una reserva de memoria dinámica de un tamaño superior a la memoria disponible empleando el operador new. Sucederá un error que hará a que termine el programa sin que podamos gestionar de ningún modo este fallo en nuestro código fuente; esto es, no se nos va a dar la opción a comprobar un valor de retorno.

La gestión de errores en base a códigos de retorno es, sin duda, insuficiente en muchos escenarios. De ahí que en Python se creasen las excepciones. Las excepciones son objetos que se "lanzan" desde los métodos o constructores para indicar que algo ha ido mal. Para lanzar excepciones se emplea la palabra reservada raise. Por ejemplo:

```
if (condiciónError) raise Exception
```

En cuanto se lanza la excepción se termina la ejecución del método y se devuelve el control al código que lo invocó. En Python cuando se invoca a un método pueden suceder dos cosas: que se ejecute correctamente y, posiblemente, nos devuelva un valor o que se lance una excepción.

Cuando se invoca a una función que puede lanzar excepciones podemos hacer dos cosas con las excepciones: gestionarlas o ignorarlas. **Si una excepción se ignora produce la inmediata terminación del método que ha ignorado la excepción y ésta continúa "escalando" en el stack de llamadas a métodos**. **Si la excepción recorre todo el stack sucederá un error en el proceso en el cual se ejecutaba el programa y el sistema operativo lo terminará**. **Si en algún momento una función gestiona el error el programador tiene la posibilidad de solucionarlo** o bien arreglando el problema que se ha producido o, al menos, mostrando un mensaje de error al usuario.

Cuando sabemos que en un fragmento de código se puede producir un error y queremos gestionarlo debemos rodear dicho fragmento de código con un bloque try-catch:

```
try: // bloque de código en el cual se puede lanzar una
  LANZA excepción
  ...
```

```
except Exception:
  MANEJO DE EXCEPCIONES
  ...
```

El código que puede generar excepciones debe ir dentro del bloque try. **Si se produce una excepción en cualquier línea de ese bloque se detiene inmediatamente la ejecución y se pasa control al bloque catch**. Entre los paréntesis del bloque catch debe indicarse el tipo de dato al que pertenece la excepción que se desea capturar. En un mismo bloque de código try podrían lanzarse muchos tipos diferentes de excepciones y puede que en el bloque catch correspondiente sólo se pueda gestionar un subconjunto de esas excepciones.

Es posible que un único bloque try tenga varios manejadores para poder gestionar distintos tipos de excepciones. Si se produce una excepción en ese bloque se buscará el manejador más adecuado para el tipo de dato de la excepción generada.

Los bloques try-except pueden llevar también un bloque finally; en este caso el código de este último bloque se ejecutará tanto si el código del try se ha ejecutado correctamente, como si se ha lanzado una excepción y nos hemos ido al catch. Los bloques finally se emplean a menudo para ejecutar código que debe ejecutarse tanto por el camino de ejecución normal como excepcional. Por ejemplo, código que cierre un archivo que hemos abierto para escritura; tanto si se produce algún error al escribir como si no tendremos que cerrarlo.

```
try: // bloque de código en el cual se puede lanzar una
  LANZA excepción

except: // bloque que maneja las posibles excepciones
  MANEJO EXCEPCIÓN
  ...
 Finally:
  CODIGO QUE SE EJECUTA SIEMPRE, HAYA SALTADO EXCEPCION O NO
  ...
```

Algunas excepciones, que se denominan uncheked, no tienen que gestionarse de modo necesario, y el compilador no nos obliga a gestionarlas, aunque sea posible que en nuestro código se produzcan excepciones de este tipo. Un ejemplo es la NoneTypeException. **Hay otras excepciones, que se denominan checked, que suelen representar problemas con el hardware que se considera que el programador debe siempre tener en mente cuando está construyendo su programa**. Estas excepciones estamos obligados a gestionarlas; las unchecked podemos gestionarlas si queremos o si no podemos ignorarlas. Cuando invoquemos a código (constructores o métodos) que potencialmente pueden lanzar una excepción cheked debemos colocar ese código dentro de un bloque try-except. Para construir una excepción checked haremos que nuestra excepción herede de Exception.

```
class GuerrerosTypeError(Exception):
  # CREATE THE OBJECTS WE WANT OR MODIFY THE DEFAULT EXCEPTION
  # or pass
```

5 Ejercicios

1. **(10 min)** Escribe un programa que cree una clase para representar un objeto punto en tres dimensiones. Proporcionar un constructor que inicialice los valores del punto al origen de coordenadas y otro que permita especificar las coordenadas del punto. Sobrescribe su método __str__ para que muestre información sobre los puntos. Usa la clase en un programa donde crees objetos que representen los puntos (12, 13, 18) y (8, 14, 0) y los muestres por consola.
2. **(30 min)** Crear una clase que represente un número racional con un numerador y un denominador guardadas en dos variables de tipo entero. Esta clase debe permitir, al menos, sumar, multiplicar y simplificar números racionales. Proporcionar un constructor por defecto, un constructor de copia (esto es, un constructor al que se le pasa una instancia de la clase número racional y crea otro número racional idéntico). Esto es, dependiendo del tipo de objeto que reciba el constructor, hará una cosa u otra. Y otro constructor que permita indicar los valores del numerador y del denominador. Usando esta clase, crea una calculadora que permita operar con números racionales, seleccionando las operaciones de un menú.
3. **(10 min)** Repetir el ejercicio anterior, pero creando una clase que represente a un número complejo en lugar de racional.
4. **(50 min)** Haz una clase llamada Persona con atributos privados: nombre, edad, DNI, sexo (usa una enumeración, ver EnumGuerreros.py para ver un ejemplo), peso y altura. Crea métodos (getters y setters) para acceder y modificar todos los atributos.

Por defecto, todos los atributos menos el DNI tendrán valores por defecto según su tipo (0 números, cadena vacía para String, etc.). Sexo será mujer por defecto.

La clase deberá tener los siguientes métodos:

- calcularIMC(): calcula el índice de masa corporal de la persona (peso en kg/(altura^2 en m))
- valorarPesoCorporal() devuelve un -1 si está por debajo de su peso ideal, un 0 si está en su peso ideal y un 1 si tiene sobrepeso. Sobrepeso se define como IMC > 25 y se considera que se está por debajo del peso ideal si IMC <18.
- esMayorDeEdad(): indica si es mayor de edad, devuelve un booleano.
- __str__() devuelve toda la información de la persona como una cadena de caracteres.
- generaDNI(): genera un numero aleatorio de 8 cifras que será el DNI de la persona. Este método no será visible desde el exterior. Este método deberá invocarse desde cualquier constructor para generar el DNI.
- Métodos set y get de cada parámetro, excepto de DNI, que sólo tendrá get

Ahora, crea una clase ejecutable que haga lo siguiente:

- Pide por teclado el nombre, la edad, sexo, peso y altura.
- Crea 3 objetos de la clase anterior, el primer objeto obtendrá las anteriores variables pedidas por teclado, el segundo objeto obtendrá todos los anteriores menos el peso y la altura y el último por defecto, para este último utiliza los métodos set para darle a los atributos un valor.
- Para cada objeto, se deberá comprobar si está en su peso ideal, tiene sobrepeso o por debajo de su peso ideal con un mensaje.
- Indicar para cada objeto si es mayor de edad.
- Por último, mostrar la información de cada objeto.

5. **(15 min)** Haz una clase llamada Password que tenga los atributos longitud y contraseña. Por defecto, la longitud será de 8. Los constructores serán los siguiente:

- Si la longitud del constructor es 0, el objeto tendrá como contraseña "password".
- Si la longitud es > 0, generará una contraseña aleatoria con esa longitud.

Los métodos de esta clase serán:

- esFuerte(): devuelve un booleano si es fuerte o no, para que sea fuerte debe tener más de 2 mayúsculas, más de 1 minúscula y más de 5 números.
- generarPassword(): genera la contraseña del objeto con la longitud que tenga.

- Método get para contraseña y longitud.
- Método set para longitud.

Ahora, crea una clase main:

- Cree una lista de Passwords con el tamaño que tú le indiques por teclado.
- Cree un bucle que cree un objeto para cada posición en la lista. Indica por teclado la longitud de cada password.
- Cree otra lista de booleanos donde se almacene si el password del array de Password es o no fuerte (usa el bucle anterior).
- Al final, muestra la contraseña y si es o no fuerte (usa el bucle anterior). Usa un diccionario con key = contraseña, value = boolean indicando si es fuerte o no.

contraseña1 valor_booleano1

contraseña2 valor_bololeano2

6. **(60 min porque implica comprender todo el código y ver las pequeñas modificaciones que hay que hacer)** Usando las clases del código de ejemplo de los marcianos, construye una guerra donde combatan 5 naves de los marcianos y 10 naves de los terrícolas.
7. **(20 min)** En una empresa todos los trabajadores tienen un sueldo base de 1000 €. Los jefes tienen un suplemento de 500 € por cada año que hayan sido jefe de la empresa, y los viajantes además del sueldo base cobran 300 € por viaje realizado. Crear una clase empleado de la cual deriven las clases jefe y viajante. Crear una plantilla de una empresa con dos jefes, cinco viajantes y 15 empleados, e imprimir por consola sus respectivos salarios. Para generar el número de viajes de los viajantes y la antigüedad de los jefes puedes generar números aleatorios entre 0 y 10. Emplea el polimorfismo de herencia.
8. **(30 min)** Crea la clase cuenta bancaria, que deberá tener como atributos un nombre de titular, una fecha de apertura, un número de cuenta y un saldo (puedes emplear un número real. La cuenta deberá tener un método para retirar dinero, otro para ingresar dinero y otro para transferir dinero a otra cuenta. De una cuenta no se podrá retirar nunca dinero (ni transferir) si la cantidad de dinero a retirar o transferir es mayor que el saldo. Crea una cuenta a plazo fijo, en la cual cuando se retira dinero de algún modo antes de

una fecha de vencimiento (que será otro atributo de esta clase) además del dinero a retirar se penaliza con un 5% adicional. Crea además una cuenta Vip, que tendrá un atributo adicional que es el saldo negativo máximo que puede tener. En las cuentas Vip uno podrá tener saldo negativo siempre que no supere este valor. A continuación, construye un main que permita crear los tres tipos de cuentas, y transferir dinero de unas a otras, ingresar dinero y retirar dinero. Almacena las cuentas en una lista. Emplea polimorfismo de herencia.

9. **(15 min)** Modifica una clase Fecha proporcionada. La clase Fecha ya almacena el día, el mes y el año de una fecha. Proporciona funciones miembro para acceder a estos atributos (getDia(), getMes() y getAño() y para modificarlos (setDia(int dia), setMes(int mes) y setAño(int año)). Sobreescribe su método __str__. Crea las excepciones correspondientes para que la clase deba asegurarse de que los valores introducidos para sus miembros, tanto a través de los constructores como de los métodos modificadores, se corresponden con una fecha válida (no es necesario tener en cuenta años bisiestos). Para ello lanzará una excepción en caso de que los datos no sean válidos. Crea la fecha 31/02/2015 y verifica que se lanza la excepción correspondiente. Verifica que esto también sucede al invocar el método setDia (35). Se debe crear una excepción propia llamada DateError.