Clases y Objetos

En un *objeto* se encapsula la *información* y las *operaciones* necesarias para operar dicho objeto.

Por ejemplo, la información sobre un coche concreto registra su estado (posición, orientación, velocidad, etc.) más las posibles operaciones (encender el motor o apagarlo, acelerar o frenar, girar, saber la velocidad, etc.)

Otro ejemplo: la información sobre una televisión incluye su estado (encendida o apagada, canal seleccionado, volumen, etc.), y las operaciones posibles, encenderla o apagarla, cambiar el volumen o el canal, consultar el volumen o el canal, etc.

Ejemplo: la clase Punto

Deseamos trabajar con objetos geométricos, y el más sencillo es el punto. Empezamos definiendo un punto en dos dimensiones; el tipo de datos correspondiente tiene sus dos coordenadas típicas.

La manera de hacerlo en Python es la siguiente:

In [1]:

```
class Point(object):
    def __init__(self):
        self.x = 0.0
        self.y = 0.0
```

In [2]:

```
p0 = Point()
p1 = Point()
p1.x, p1.y = 4., 5.
print (p0.x, p0.y)
print (p1.x, p1.y)
print(type(p0))
```

```
0.0 0.0
4.0 5.0
<class '__main__.Point'>
```

Hemos definido la *clase* `Punto`, que es un objeto genérico, y esta definición nos ha permitido luego crear los *objetos* particulares `p0` y `p1`, variables que representan puntos concretos. Este objeto únicamente tiene de momento información, las coordenadas `x` e `y`. Estos datos definen la posición del punto, su *estado*, y cada uno de ellos es un *atributo* de la clase `Punto`.

Para referirnos a los atributos, dentro de la clase, usamos el identificador `self`.

Vamos a añadir una operación, para saber la distancia del punto al origen de coordenadas:

In [3]:

```
from math import sqrt, pi

class Point(object):
    def __init__(self):
        self.x = 0.0
        self.y = 0.0
    def dist_origen(self):
        return sqrt(self.x**2 + self.y**2)

p = Point()
p.x, p.y = 12.0, 5.0
print(p.dist_origen())
```

```
13.0
```

Acabamos de definir la primera operación, que se llama *método* en el mundo de la Programación Orientada a Objetos (POO).

Ejemplo de uso de estos objetos. Podemos escribir una función que calcule la distancia entre dos Points:

In [4]:

```
from math import sqrt, pi

def distancia(p0, p1):
    return sqrt((p0.x - p1.x)**2 + (p0.y - p1.y)**2)
```

Con una función como ésta, es fácil discernir si cuatro Points forman un rectángulo:

In [5]:

```
def es_rectangulo(a, b, c, d):
    dab = distancia(a, b)
    dac = distancia(a, c)
    dad = distancia(a, d)
    dbc = distancia(b, c)
    dbd = distancia(b, d)
    dcd = distancia(c, d)
    return dab == dcd and dac == dbd and dad == dbc

p0, p1, p2, p3 = Point(), Point(), Point(), Point()

p0.x, p0.y = 0, 0
p1.x, p1.y = 1, 1
p2.x, p2.y = 0, 1
p3.x, p3.y = 1, 0

es_rectangulo(p0, p1, p2, p3)
```

Out[5]:

```
True
```

Nota: En realidad, la distancia entre Points debería definirse mejor como un *método* de la clase Point. Lo

veremos más adelante.

Métodos especiales __init__ y __str__

La definición anterior no permite definir más puntos que el (0, 0). Aunque luego se pueda cambiar, sería mejor poder definir un punto que, inicialmente, tenga la posición que se desee.

Para ello, se debe utilizar el *constructor* `__init__` , que es un método especializado en crear objetos con un determinado estado:

In [6]:

```
class Point(object):
    """
    Point class. It represents 2D points.

    Attributes
    ---------
    x, y: float
    """
    def __init__(self, px, py):
        """
        Constructor

        Parameters
        ----------
        x: float
        y: float
        """
        self.x = px
        self.y = py
```

In [7]:

```
p0, p1, p2, p3 = Point(0,0), Point(3,1), Point(0,1), Point(3,0)
print(distancia(p0, p1))
print(es_rectangulo(p0, p1, p2, p3))
```

```
3.1622776601683795
True
```

In [8]:

```
print(p0)
p0
```

```
<__main__.Point object at 0x0000029DCA3E3A58>
```

Out[8]:

```
<__main__.Point at 0x29dca3e3a58>
```

Otro método especial es `__str__` que determina cómo se escriben (con `print`) los objetos de una clase:

In [9]:

```
class Point(object):
    """
    clase Point. Representa  puntos en 2D

    Attributes
    ----------
    x, y: float
    """
    def __init__(self, px, py):
        """
        Constructor

        Parameters
        ----------
        x: float
        y: float
        """
        self.x = px
        self.y = py

    def __str__(self):
        """
        Este metodo devuelve el str que representa un Point
        """
        return '({0:.2f}, {1:.2f})'.format(self.x, self.y)
```

In [10]:

```
p0 = Point(3.0, 4.0)
print(p0)
p0
```

```
(3.00, 4.00)
```

Out[10]:

```
<__main__.Point at 0x29dca2ecb00>
```

Más métodos

También podemos *encapsular* la función distancia dentro de la clase `Point`

In [11]:

```
class Point(object):
    def __init__(self, px, py):
        self.x = px
        self.y = py

    def __str__(self):
        return '(' + str(self.x) + ', ' + str(self.y) + ')'

    def distance(self, other):
        """
        This function returns the distance from this object to other
        """
        return sqrt((self.x - other.x)**2 + (self.y - other.y)**2)
```

In [12]:

```
p = Point(5, 6)
q = Point(6, 7)
print(p, q, p.distance(q))
```

```
(5, 6) (6, 7) 1.4142135623730951
```

In [13]:

```
def es_rectangle(a, b, c, d):
    dab = a.distance(b)
    dac = a.distance(c)
    dad = a.distance(d)
    dbc = b.distance(c)
    dbd = b.distance(d)
    dcd = c.distance(d)
    return dab == dcd and dac == dbd and dad == dbc

p0, p1, p2, p3 = Point(0,0), Point(1,1), Point(0,1), Point(1,0)
es_rectangle(p0, p1, p2, p3)
```

Out[13]:

```
True
```

También hay métodos que modifican el estado del objeto. Pensemos, por ejemplo, en mover el objeto.

In [14]:

```
class Point(object):
    def __init__(self, px, py):
        self.x = px
        self.y = py

    def __str__(self):
        return 'Point(' + str(self.x) + ', ' + str(self.y) + ')'

    def distance(self, other):
        return sqrt((self.x - other.x)**2 + (self.y - other.y)**2)

    def move(self, t_x, t_y):
        self.x = self.x + t_x
        self.y = self.y + t_y
```

Observa que el método `move` no tiene `return`, modifica la posición de un Point pero no devuelve nada.

In [15]:

```
p0 = Point(1.0, 2.0)
p1 = Point(7.0, 3.5)
print(p0)
print(p1)
print(p0.distance(p1))
p0.move(2.0, 4.0)
print(p0)
```

```
Point(1.0, 2.0)
Point(7.0, 3.5)
6.18465843842649
Point(3.0, 6.0)
```

Si definimos el objeto Vector, un punto se puede mover según un vector, más cómodamente.

In [16]:

```
class Point(object):
    def __init__(self, px, py):
        self.x = px
        self.y = py

    def __str__(self):
        return 'Point({0:.2f}, {1:.2f})'.format(self.x, self.y)

    def distance(self, other):
        return sqrt((self.x - other.x)**2 + (self.y - other.y)**2)

    def move(self, v):
        """
        This function move this point applying the vector v
        """
        self.x = self.x + v.x
        self.y = self.y + v.y

class Vector(object):
    """This class reperesents a 2D vector

    Attributes
    ---------
    x, y: float
    """
    def __init__(self, px, py):
        """
        Constructor

        Parameters
        ----------
        x: float
        y: float
        """
        self.x = px
        self.y = py

    def __str__(self):
        """
        This method returns str representation of the Vector
        """
        return 'Vector({0:.2f}, {0:.2f})'.format(self.x, self.y)
```

In [17]:

```
p0 = Point(1.0, 3.0)
print(p0)
p0.move(Vector(2.0, 1.0))
print(p0)
```

```
Point(1.00, 3.00)
Point(3.00, 4.00)
```

Objetos como atributos de otro objeto

In [18]:

```
class Circle(object):
    """
        Class to represents a circle.
    """
    def __init__(self, center, radius):
        """
        Constuctor

        Parameters:
        -----------
        center: Point
        radius: float
        """
        self.center = center
        self.radius = radius

    def __str__(self):
        return 'Circle({0}, {1:.2f})'.format(self.center, self.radius)

    def surface(self):
        """
        This function returns the surface of the circle
        """
        return pi*self.radius**2
```

In [19]:

```
p0 = Point(2.0, 1.0)
c = Circle(p0, 2.0)
print(c)
print(c.surface())
```

```
Circle(Point(2.00, 1.00), 2.00)
12.566370614359172
```

Y si sabemos trasladar un Point, sabemos trasladar un círculo:

In [20]:

```
class Circle(object):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    def __str__(self):
        return 'Circle({0}, {1:.2f})'.format(self.center, self.radius)

    def surface(self):
        """
        This function returns the surface of the circle
        """
        return pi*self.radius**2

    def move(self, v):
        """
        This funciton move the Circle applying vector v
        """
        self.center.move(v)
```

In [21]:

```
p0 = Point(2.0, 1.0)
c = Circle(p0, 2.0)
print(c)
c.move(Vector(3.0, 1.0))
print(c)
print(p0)
```

```
Circle(Point(2.00, 1.00), 2.00)
Circle(Point(5.00, 2.00), 2.00)
Point(5.00, 2.00)
```

Observemos que se ha producido un efecto lateral. Al mover el círculo, se ha movido también el punto. Se comparte memoria. Si no quiero que pase tengo que hacer una copia.

In [22]:

```
from copy import deepcopy as dcopy

p0 = Point(2.0, 1.0)
c = Circle(dcopy(p0), 2.0)
print(c)
c.move(Vector(3.0, 1.0))
print(c)
print(p0)
```

```
Circle(Point(2.00, 1.00), 2.00)
Circle(Point(5.00, 2.00), 2.00)
Point(2.00, 1.00)
```

Métodos especiales

En todas las clases se pueden definir métodos con nombres concretos para poder usar las clases de forma más cómoda: https://docs.python.org/3/reference/datamodel.html#special-method-names (https://docs.python.org/3/reference/datamodel.html#special-method-names)

In [23]:

```
class Point(object):
    def __init__(self, px, py):
        self.x = px
        self.y = py

    def __str__(self):
        return 'Point({0:.2f}, {1:.2f})'.format(self.x, self.y)


    def __add__(self, v):
        """
        This function returns a new point adding vector v to self

        Paramters
        ---------
        v: Vector

        Returns
        -------
        Point
        """
        if not isinstance(v, Vector):
            return NotImplemented
        else:
            return Point(self.x + v.x, self.y + v.y)


    def __sub__(self, p):
        """This method returns the vector form p to self

        Paramters
        ---------
        p: Point

        Returns
        -------

        Vector
        """
        if not isinstance(p, Point):
            return NotImplemented
        else:
            return Vector(self.x - p.x, self.y - p.y)


    def distance(self, p):
        """This method computes the distance from shelf to p

        Parameters
        ----------
        p: Point

        Returns
        -------
        float
        """
        return (self - p).module()

class Vector(object):
```

```
    """This class reperesents a 2D vector

    Attributes
    ---------
    x,y: float
    """
    def __init__(self, px, py):
        """
        Constructor

        Parameters
        ----------
        x: float
        y: float
        """
        self.x = px
        self.y = py

    def module(self):
        """This method returns the module of a vector"""
        return sqrt(self.x**2 + self.y**2)

    def __str__(self):
        """
        This method returns str representation of the Vector
        """
        return 'Vector({0:.2f}, {0:.2f})'.format(self.x, self.y)
```

In [24]:

```
p = Point(0,1)
v = Vector(2,2)
q = p + v
q.distance(p)
```

Out[24]:

```
2.8284271247461903
```

Ejemplo adicional

... para una agenda... a lo mejor puedes completar tú los fragmentos que faltan...

In [1]:

```
class Persona(object):
    """
    Esta clase representa la ficua de una persona.
    No incluye el núm. de registro o el DNI,
    porque se asume que puede ser la clave de búsqueda
    en un diccionario.

    Attributes
    ---------
    nombre: string
    edad: int
    estatura: float
    direccion: string
    """

    def __init__(self, nombre, edad, estatura, direccion):
        self.nombre = nombre
        self.edad = edad
        self.estatura = estatura
        self.direccion = direccion

    def __str__(self):
        #...
        return '<Nombre: ' + self.nombre + ', Edad: ' + str(self.edad) + ', ...>'

p = Persona("Blacky", 12, 0.30, "Carretera de Húmera, Pozuelo de Alarcón")
print(p)
p
```

```
<Nombre: Blacky, Edad: 12, ...>
```

Out[1]:

```
<__main__.Persona at 0x185b0ca69a0>
```

In [24]:

```
# Formamos ahora una agenda con un diccionario, donde la clave es el número de registro:

mi_agenda = dict()
mi_agenda["7023"] = p

def mostrar_agenda(agenda):
    for n in agenda:
        print(n, agenda[n])

mostrar_agenda(mi_agenda)
```

```
7023 <Nombre: Blacky, Edad: 12, ...>
```

In [40]:

```
# Leemos el contenido de un archivo en la agenda

def crear_agenda(nombre_archivo):
    la_agenda = dict()
    archivo = open(nombre_archivo, "r")
    for linea in archivo:
        # print(linea) # just for testing
        lin_limpia = linea.rstrip('\n')
        reg, nom, edad, estat, direcc = lin_limpia.split(" # ")
        edad = int(edad)
        estat = float(estat)
        # print(reg, nom, edad, estat, direcc) # just for testing
        la_agenda[reg] = Persona(nom, edad, estat, direcc)
    return la_agenda

ag = crear_agenda("agenda.txt")
mostrar_agenda(ag)
print(ag["023491"].direccion)
```

```
023491 <Nombre: Fernando, Edad: 22, ...>
324098 <Nombre: Elena   , Edad: 56, ...>
Pozuelo de Alarcón
```

In [2]:

```
# Definimos una clase extendiendo la anterior: una clase derivada:

class Familiar(Persona):

    def __init__(self, nombre, edad, estatura, direccion, par):
        Persona.__init__(self, nombre, edad, estatura, direccion)
        self.parentesco = par

    def __str__(self):
        return '<Nombre: ' + self.nombre + ', Edad: ' + str(self.edad) \
            + ', "Parentesco: ' + str(self.parentesco) + '>'

    def es_humano(self):
        if self.parentesco == "Mascota":
            return False
        else:
            return True

f = Familiar("Blacky", 12, 0.30, \
             "Carretera de Húmera, Pozuelo de Alarcón", "Mascota")

print(f)

print(f.es_humano)
```

```
<Nombre: Blacky, Edad: 12, "Parentesco: Mascota>
```

In [3]:

```
# Definimos una clase extendiendo la anterior: una clase derivada:

class Familiar(Persona):

    def __init__(self, nombre, edad, estatura, direccion, par):
        Persona.__init__(self, nombre, edad, estatura, direccion)
        self.parentesco = par

    def __str__(self):
        return '<Nombre: ' + self.nombre + ', Edad: ' + str(self.edad) \
            + ', "Parentesco: ' + str(self.parentesco) + '>'

    def es_humano(self):
        if self.parentesco == "Mascota":
            return False
        else:
            return True
```

In [5]:

```
# He quí unas pocas pruebas:

f = Familiar("Blacky", 12, 0.30, \
             "Carretera de Húmera, Pozuelo de Alarcón", "Mascota")

print(f)

print(type(f))

print(f.es_humano)
```

```
<Nombre: Blacky, Edad: 12, "Parentesco: Mascota>
<class '__main__.Familiar'>
<bound method Familiar.es_humano of <__main__.Familiar object at 0x00000185B
0CA6370>>
```

In [ ]:

```

```