Definiendo nuevos tipos de datos: Clases

Python nos permite crear nuevos tipos de datos para guardar información de manera más sofisticada.

Por ejemplo, supongamos que queremos trabajar con objetos geométricos.

De ellos los más básicos son los Points, empezaremos en dos dimensiones.

Para ello necesitamos definir un nuevo tipo de datos al que llamamos `Point`. Para definir un tipo de datos utilizamos la sintaxis siguiente:

In [9]:

```
from math import sqrt, pi
```

In [10]:

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

El identificador `Point` es el nombre que utilizaremos luego para definir objetos de este tipo. Las variables `x` e `y` almacenan el estado del objeto, en este caso un Point en dos dimensiones.

Pero, si estamos definiendo objetos es porque queremos utilizarlos ...

In [11]:

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

```
0.0 0.0
4.0 5.0
```

In [12]:

```
p0, str(p0), p0.x, p0.y, type(p0)
```

Out[12]:

```
(<__main__.Point at 0x7f3ac81ba588>,
 '<__main__.Point object at 0x7f3ac81ba588>',
 0.0,
 0.0,
 __main__.Point)
```

In [13]:

```
print(p0)
```

```
<__main__.Point object at 0x7f3ac81ba588>
```

Los Points que hemos definido tienen dos *atributos* que son sus coordenadas. Podemos escribir una función que nos devuelva la distancia entre dos Points

In [14]:

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

De esta forma podemos escribir mucho mejor el ejercicio de la hoja 1 en que debíamos discernir si cuatro Points formaban un rectángulo:

In [15]:

```
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[15]:

```
True
```

Nota: La distancia entre Points debe ser un *método* de la clase Point. Lo veremos más adelante.

Métodos especiales __init__ y __str__

Lo que hemos visto anteriormente no es la forma correcta de definir *objetos* de una *clase*. La forma correcta de inicializar los distintos Points es utilizar lo que se denomina un *constructor*. El método `__init__` se encarga de crear objetos con un determinado estado:

In [16]:

```
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 [17]:

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

Out[17]:

```
True
```

In [18]:

```
print(p0)
p0
```

```
<__main__.Point object at 0x7f3ac81ba898>
```

Out[18]:

```
<__main__.Point at 0x7f3ac81ba898>
```

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

In [19]:

```
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

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

In [20]:

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

```
(3.00, 3.00)
```

Out[20]:

```
<__main__.Point at 0x7f3ac81ba710>
```

In [21]:

```
p1 = Point(6.0, 0.0)
distancia(p0, p1)
```

Out[21]:

```
5.0
```

Definiendo otros métodos

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

In [22]:

```
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 [23]:

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

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

In [24]:

```
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[24]:

```
True
```

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

In [25]:

```
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 `traslada` no tiene `return`, se encarga de hacer el movimiento en el Point pero no devuelve nada.

In [26]:

```
p0 = Point(1.0, 2.0)
print(p0)
p0.move(2.0, 4.0)
print(p0)
```

```
Point(1.0, 2.0)
Point(3.0, 6.0)
```

In [27]:

```
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 [28]:

```
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 [29]:

```
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 [30]:

```
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 [31]:

```
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 [32]:

```
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 [33]:

```
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 [36]:

```
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 [37]:

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

Out[37]:

2.8284271247461903

In [ ]:

```

```