GOBIERNO DE ESPAÑA
MINISTERIO DE HACIENDA Y FUNCIÓN PÚBLICA
INAP
INSTITUTO NACIONAL DE ADMINISTRACIÓN PÚBLICA

Introducción a la programación en Python

Tema 7. Programación Orientada a Objetos

Autor: Francisco Mariño Ruiz

Contenido

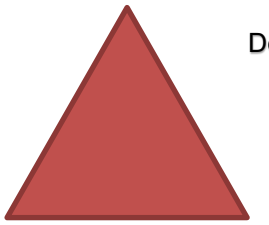
1 Clases y objetos

1.1 Introducción

Los objetos en programación son una manera de organizar la información y de abstraer el mundo real al mundo de la programación. Los objetos serán representaciones de entidades que contienen una serie de propiedades y de funciones propias.

Desde el punto de vista formal, deberíamos distinguir tres conceptos clave:

- Objeto: Elemento del mundo real descrito con propiedades y funciones.
- Clase: Definición formal de un objeto.
- Instancia: Variables que generamos a partir de una determinada clase.



Definición de clase

desde objeto

```
class Triangulo(object):

    def __init__(self, Punto1, Punto2, Punto3):
        self.Punto1 = Punto1
        self.Punto2 = Punto2
        self.Punto2 = Punto3

    def perimetro(self):
        # Cuerpo de la función

    def area(self):c
        # Cuerpo de la función
```

Instancia

desde clase

```
# Nuevo Triángulo
triangulo = Triangulo(Punto1, Punto2, Punto3)
```

En el esquema anterior podemos ver el siguiente ejemplo. Tenemos un objeto de la realidad que es un triángulo, este objeto tiene diversas propiedades, como son sus lados, sus ángulos o sus coordenadas. Para definir una clase desde un objeto de la realidad podemos hacer uso de sus propiedades, utilizando sólo las que consideremos importantes para nuestro proyecto y de una serie de funciones propias de nuestro objeto como son en este caso la función que calcula el área y la que calcula el perímetro. Cada vez que generemos un nuevo triangulo en nuestro programa usando la clase que hemos definido previamente, estaremos generando una instancia de esa clase. En Python todos los tipos de datos están definidos como objetos con sus respectivas funciones, lo cual es característico de Python, ya que no es común a todos los lenguajes de programación.

1.2 Definiendo nuevos tipos

Aunque Python nos proporciona gran cantidad de tipos ya definidos, en numerosas ocasiones necesitaremos utilizar otros tipos distintos a los ya definidos, será entonces cuando crearemos nuestros propios tipos, que almacenarán la información que necesitemos para resolver un problema y contendrán las funciones que necesitemos para manejar dicha información.

Para desarrollar la creación de nuevos tipos, vamos a implementar la clase "punto". Esta clase pretende representar puntos en un plano a través de sus coordenadas cartesianas. En primer lugar, vamos a crear la clase donde sólo se almacenen las coordenadas.

```
1  class Punto(object):
2  """Representa un punto sobre un plano con sus coordenadas cartesianas"""
3
4      def __init__(self, x=0, y=0):
5          """Constructor del punto"""
6          self.x = x
7          self.y = y
```

En la primera línea de código indicamos que vamos a crear una nueva clase llamada Punto. La palabra *"object"* entre paréntesis indica que la clase es un objeto básico no basada en ningún objeto más complejo.

En la cuarta línea hemos usado el método especial *"__init__"* que es el constructor de clase y que es el método que se usa cada vez que se crea una nueva instancia de esa clase. Los parámetros que se le pasan son la palabra reservada *"self"* que hace referencia a la propia instancia que se está creando, más los atributos que queremos generar.

Para asignar los valores dentro del constructor, se distinguen las propiedades del objeto de los parámetros de entrada de la función utilizando de nuevo la palabra clave *"self"* seguida de un punto y el nombre del atributo que estamos asignando. Se recomienda que todos los atributos de un objeto se definan dentro de la función constructor.

Además, hemos añadido a continuación de la definición de la clase y de la definición de la primera función una cadena de texto entre comillas triples. Esta cadena de texto es totalmente opcional, aunque recomendable cuando trabajemos en proyectos con terceros, está definida para ser una pequeña ayuda que explique a un usuario para qué sirve la función. Se utiliza con el comando "*help()*" desde consola.

```
>>> help (Punto)
Help on class Punto in module Punto:
class Punto(builtins.object)
 |  Representa un punto sobre un plano con sus coordenadas cartesianas
 |
 |  Methods defined here:
 |
 |  __init__(self, x=0, y=0)
 |      Constructor del punto
```

```
>>> help (Punto.__init__)
Help on function __init__ in module Punto:
__init__(self, x=0, y=0)
    Constructor del punto
```

Para utilizar el objeto que hemos creado como ejemplo, bastaría con usar el siguiente código (si lo escribimos en un módulo distinto, hay que importar el módulo donde hayamos definido el objeto):

```
1  p = Punto(5, 7)
2
3  print(p)
4  print(p.x)
5  print(p.y)
```

```
<__main__.Punto object at 0x00000249B46A8160>
5
7

Process finished with exit code 0
```

Al realizar la llamada de la línea 1, se crea una nueva instancia de la clase Punto y se almacena en la variable *"p"*, asignándose 5 y 7 a los valores de las propiedades *"x"* e *"y"*. Aunque nosotros no hayamos llamado explícitamente a la función del constructor, internamente Python se ha encargado de hacerlo y asignar los valores de sus propiedades.

Al crear la clase *"Punto"* permitimos guardar dos valores *"x"* e *"y"*, sin embargo, aunque documentemos nuestro objeto e indiquemos a los posibles usuarios que los valores deben ser numéricos, el código mostrado no impide que se les asigne cualquier valor no numérico.

```
1  q = Punto("A", True)
2
3  print(q)
4  print(q.x)
5  print(q.y)
```

```
<__main__.Punto object at 0x000001FD938C8940>
```

```
A
True

Process finished with exit code 0
```

Esto puede ser potencialmente problemático a la hora de operar con los datos en funciones posteriores. Para evitar que esto suceda, vamos a forzar que los datos de entrada sean convertidos a tipo *"float"*, esto nos garantizará que solo si introducimos un dato numérico se creará el punto, en caso contrario nos devolverá un error.

```
1  class Punto(object):
2  """Representa un punto sobre un plano con sus coordenadas cartesianas"""
3
4      def __init__(self, x=0, y=0):
5              self.x = float(x)
6              self.y = float(y)
7
8  q = Punto("A", True)
```

```
Traceback (most recent call last):
  File "C:/Users/fmari/OneDrive/Datos/CursoPython18/Ejemplos/Punto.py", line 8, in
<module>
    t = Punto("a", True)
  File "C:/Users/fmari/OneDrive/Datos/CursoPython18/Ejemplos/Punto.py", line 5, in
__init__
    self.x = float(x)
ValueError: could not convert string to float: 'A'

Process finished with exit code 1
```

1.3 Agregando funciones a nuestras clases

Hasta ahora la clase *"Punto"* solo es un contenedor de información con un par de valores que hemos establecido que siempre serán numéricos. Sin embargo, para explotar completamente la funcionalidad de los objetos podemos definir funciones a los mismos para operar con su contenido.

Vamos a crear una primera función que nos devuelva el punto formateado en una cadena de texto que representará sus coordenadas. La función no recibirá ningún parámetro y nos devolverá una cadena de texto. Las funciones de cada clase se definen dentro de la definición de la clase:

```
1  class Punto(object):
2  """Representa un punto sobre un plano con sus coordenadas cartesianas"""
3
4      def __init__(self, x=0, y=0):
5              self.x = float(x)
6              self.y = float(y)
7
8      def punto_to_string(self):
9          return "x = {}, y = {}".format(self.x, self.y)
```

Podremos utilizarla sobre cualquier instancia de *"Punto"* que definamos.

```
1  p = Punto(5.0, 7.2)
2
3  print(p.punto_to_string())
```

```
x = 5.0, y = 7.2

Process finished with exit code 0
```

A continuación, vamos a crear una función con parámetros. En este caso vamos a crear una función que reciba como parámetro un segundo punto y calcule la distancia entre ambos.

```
18     def distancia(self, otro_punto):
19         dx = self.x - otro_punto.x
20         dy = self.y - otro_punto.y
21         dist = (dx**2 + dy**2)**0.5
22         return dist
```

Y la usaríamos del siguiente modo:

```
1  p = Punto(5.0, 7.2)
2  q = Punto(1.4, 9.1)
3
4  print(p.distancia(q))
```

```
5.091168824543142

Process finished with exit code 0
```

Vamos a definir una tercera función que nos calcule el acimut entre esos dos puntos, además en esta función vamos a necesitar funciones de un módulo externo, que necesitaremos importar, el módulo numpy.

```
25      def acimut(self, otro_punto):
26          dx = self.x - otro_punto.x
27          dy = self.y - otro_punto.y
28          acimut_rad = numpy.arctan2(dx, dy)
29          acimut_gra = (acimut_rad * 200) / numpy.pi
30          return (acimut_gra)
```

Y la usaríamos del siguiente modo:

```
1   p = Punto(5.0, 7.2)
2   q = Punto(1.4, 9.1)
3
4   print(p.acimut(q))
```

```
    130.9156626491703

    Process finished with exit code 0
```

Si nos fijamos en ambas funciones, ambas repiten dos instrucciones, las que obtienen los diferenciales de las coordenadas. Probablemente esta operación se repetirá en alguna función más que definamos, por lo que sería una buena práctica definir una función que calcule esa diferencia y nos la devuelva como una tupla.

```
32      def resta(self, otro_punto):
33          dx = self.x - otro_punto.x
34          dy = self.y - otro_punto.y
35          return (dx, dy)
```

Esto nos va a servir para ver cómo llamar a una función de una clase, desde dentro de la propia clase, lo haremos con la palabra clave *"self"* seguida de un punto y la llamada a la función. Nuestras funciones de distancia y acimut quedarían como vemos a continuación.

```
18      def distancia(self, otro_punto):
19          dx, dy = self.resta(otro_punto)
20          dist = (dx**2 + dx**2)**0.5
21          return dist
22
23      def acimut(self, otro_punto):
24          dx, dy = self.resta(otro_punto)
25          acimut_rad = numpy.arctan2(dx, dy)
26          acimut_gra = (acimut_rad * 200) / numpy.pi
27          return (acimut_gra)
```

Hemos creado una función para utilizar en otras funciones del mismo objeto, sin embargo, de la manera que la hemos definido dentro del objeto, nuestra función de resta puede ser llamada desde cualquier instancia de la clase punto. Como queremos que sea una función interna y que no pueda ser llamada desde fuera de la clase, la vamos a convertir en una función privada, de modo que solo será visible y usable dentro de la propia clase. Para convertir una función en privada basta con añadir dos guiones bajos delante de su nombre.

```
18      def distancia(self, otro_punto):
19          dx, dy = self.__resta(otro_punto)
20          dist = (dx**2 + dx**2)**0.5
21          return dist
22
23      def acimut(self, otro_punto):
24          dx, dy = self.__resta(otro_punto)
25          acimut_rad = numpy.arctan2(dx, dy)
26          acimut_gra = (acimut_rad * 200) / numpy.pi
27          return (acimut_gra)
28
29      def __resta(self, otro_punto):
30          dx = self.x - otro_punto.x
31          dy = self.y - otro_punto.y
32          return (dx, dy)
```

De este modo nuestras funciones de dentro de la clase la utilizarán, pero si intentamos llamarla desde fuera, obtendremos un error.

```
1  p = Punto(5.0, 7.2)
2  q = Punto(1.4, 9.1)
3
4  print(p.acimut(q))
5  print(p.acimut(q))
6  print(p.__resta(q))
```

```
5.091168824543142
130.9156626491703
Traceback (most recent call last):
  File "C:/Users/fmari/OneDrive/Datos/CursoPython18/Ejemplos/Punto.py", line 42,
in <module>
    print(p.__resta(q))
AttributeError: 'Punto' object has no attribute '__resta'

Process finished with exit code 1
```

1.4 Métodos especiales

Si recordamos el principio del capítulo, para definir el constructor de nuestra clase *"Punto"* utilizamos el método especial *"__init__"*. En Python existen diversos métodos especiales, que si definimos en nuestras clases, Python los usará cuando hagamos uso de una instancia en situaciones particulares.

La primera función que definimos, fue una función que nos formateaba el contenido de nuestro punto como texto. Sin embargo, Python nos proporciona el método especial *"__str__"* para mostrar nuestros objetos en forma de cadena, que será llamado cuando se intente hacer una conversión explícita. Modifiquemos nuestro método y probemos su comportamiento.

```
15      def __str__(self):
16          return "x = {}, y = {}".format(self.x, self.y)
```

```
42  print(p.__str__())
43  print(str(p))
44  print(p)
```

```
x = 5.0, y = 7.2
x = 5.0, y = 7.2
x = 5.0, y = 7.2

Process finished with exit code 0
```

Se puede observar que tanto si lo llamamos directamente, como si usamos el conversor explícito a *"string"* o si directamente le pedimos a Python que nos lo muestre por pantalla obtenemos el mismo resultado.

Otros métodos especiales son los siguientes:

"__len__": Nos permite establecer una función, que será el resultado cuando pidamos la longitud de una instancia con el método genérico *"len()"*. Por ejemplo, como se trata de un punto, vamos a definir que su dimensión es cero y por tanto su longitud será cero.

```
18      def __len__(self):
19          return 0
```

```
45  print(p.__len__())
46  print(len(p))
```

```
0
0

Process finished with exit code 0
```

"__add__": Nos permite establecer el comportamiento, cuando se intenten sumar dos instancias de nuestra clase. Vamos a definir la función para que nos devuelva una nueva instancia de la clase *"Punto"* con la suma de las coordenadas de ambos.

```
21      def __add__(self, otro_punto):
22          return Punto(self.x + otro_punto.x, self.y + otro_punto.y)
```

```
38  p = Punto(5.0, 7.2)
39  q = Punto(1.4, 9.1)
40  print(p + q)
```

```
x = 6.4, y = 16.3

Process finished with exit code 0
```

"__sub__": Equivale al método anterior, pero para realizar la resta.

```
25      def __sub__(self, otro_punto):
26          return Punto(self.x - otro_punto.x, self.y - otro_punto.y)
```

```
38  p = Punto(5.0, 7.2)
39  q = Punto(1.4, 9.1)
40  print(p - q)
```

```
x = 3.6, y = -1.8999999999999995

Process finished with exit code 0
```

Podríamos definir el comportamiento de la multiplicación con *"__mul__"*, el de la división con *"__div__"*, el del módulo con *"__mod__"*, el de la potencia con *"__pow__"*, etc.

Existe una gran cantidad de métodos especiales que podemos definir para nuestras clases, para conocerlos conviene echar un vistazo a la documentación oficial de Python.

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

1.5 Ejemplo de uso de clases dentro de otras clases

En los apartados anteriores hemos definido la clase *"Punto"* y algunos métodos de la misma. En los ejemplos que hemos visto hasta ahora, instanciábamos a la clase directamente y la usábamos. En este apartado veremos que es posible utilizar las clases que nosotros definamos, dentro de otras clases.

Por ejemplo, vamos a definir una clase *"Linea"*, cuya única propiedad será una lista de puntos. En su constructor estableceremos como único parámetro dicha lista y su valor por defecto será una lista vacía. Además, comprobaremos que los elementos de la lista son puntos y no otro tipo de dato utilizando la función "*isinstance()*" que nos permite comprobar si una instancia pertenece a una clase. En caso de no ser así, forzaremos un error. (Aunque la gestión de errores se verá en el tema 8 y se explicará con más detalle, en este ejemplo veremos el uso del comando *"raise"* que detiene un programa y muestra un error).

```
1  from Punto import Punto
2
3  class Linea(object):
4      """Representa una lista de puntos que forman una línea"""
5
6      def __init__(self, puntos =[]):
7          self.puntos = []
8          for punto in puntos:
9              if isinstance(punto, Punto):
10                 self.puntos.append(punto)
11             else:
12                 raise Exception('Los elementos de una línea deben de'
13                                 ' ser puntos')
```

Ya podríamos usar nuestra nueva clase.

```
1  p = Punto(5.0, 7.2)
2  q = Punto(1.4, 9.1)
3
4  lin = Linea([p, q])
5  print(lin)
```

```
<__main__.Linea object at 0x000002A4D1C88D68>

Process finished with exit code 0
```

Y en caso de que intentásemos introducir un parámetro incorrecto.

```
p = Punto(5.0, 7.2)

lin = Linea([p, “a”])
print(lin)
```

```
Traceback (most recent call last):
  File "C:/Users/fmari/OneDrive/Datos/CursoPython18/Ejemplos/Linea.py", line 19,
in <module>
    lin = Linea([p, "a"])
  File "C:/Users/fmari/OneDrive/Datos/CursoPython18/Ejemplos/Linea.py", line 12,
in __init__
    raise Exception('Los elementos de una línea deben de'
Exception: Los elementos de una línea deben de ser puntos

Process finished with exit code 1
```

Ahora vamos a definir el método especial *“__str__”* que se va a apoyar en el método del mismo nombre que creamos para la clase *“Punto”*.

```
    def __str__(self):
        cadena = ''
        for punto in self.puntos:
            if isinstance(punto, Punto):
                cadena += str(punto) + ", "
        return cadena[:-2]
```

Por lo tanto, ya podemos imprimir nuestra línea.

```
p = Punto(5.0, 7.2)
q = Punto(1.4, 9.1)

lin = Linea([p, q])
print(lin)
```

```
x = 5.0, y = 7.2, x = 1.4, y = 9.1

Process finished with exit code 0
```

Para continuar con el ejemplo, definiremos una función que permita añadir puntos a la línea. Esta función tiene como finalidad añadir precisión a nuestras líneas o extenderlas, por lo tanto, vamos a definir la función con un parámetro obligatorio, que será el punto que queremos añadir y un parámetro opcional que será la posición, siendo su valor por defecto la última posición de la lista. A esta función también añadiremos la comprobación de tipo de objeto.

```
    def addPunto(self, nuevo_punto, pos=-1):
        if pos == -1:
            pos = len(self.puntos)
        if isinstance(nuevo_punto, Punto):
           self.puntos.insert(pos, nuevo_punto)
        else:
           raise Exception ('El objeto que se intenta insertar no es un punto.')
```

Lo probamos:

```
p = Punto(5.0, 7.2)
q = Punto(1.4, 9.1)

lin = Linea([p, q])
print(lin)

s = Punto(0.0, 10.9)
lin.addPunto(s)
print(lin)

t = Punto(3.5, 8.1)
lin.addPunto(t, 1)
print(lin)
```

```
x = 5.0, y = 7.2, x = 1.4, y = 9.1
x = 5.0, y = 7.2, x = 1.4, y = 9.1, x = 0.0, y = 10.9
x = 5.0, y = 7.2, x = 3.5, y = 8.1, x = 1.4, y = 9.1, x = 0.0, y = 10.9

Process finished with exit code 0
```

Sin embargo, esta función requiere que construyamos un punto y lo añadamos, lo cual nos requiere dos pasos. Vamos a definir una función que haga lo mismo, pero solo pasándole las coordenadas y usando internamente el constructor de la clase *"Punto"*, de este modo veremos lo sencillo y útil que es mezclar nuestras clases.

```
    def addPuntoXCoordenadas(self, x, y, pos=-1):
        if pos == -1:
            pos = len(self.puntos)
        punto = Punto(x, y)
        self.puntos.insert(pos, punto)
```

Si nos fijamos, la función evita tener que hacer comprobaciones, ya que se hacen en el propio constructor de punto, por lo que nos evitamos repetir ese paso. El resultado es igual, que en la función anterior.

```
p = Punto(5.0, 7.2)
q = Punto(1.4, 9.1)

lin = Linea([p, q])
print(lin)

lin.addPuntoXCoordenadas(3.5, 8.1)
print(lin)
```

```
x = 5.0, y = 7.2, x = 1.4, y = 9.1
x = 5.0, y = 7.2, x = 1.4, y = 9.1, x = 3.5, y = 8.1

Process finished with exit code 0
```

1.6 Clases y herencia

La herencia es una característica de la programación orientada a objetos que sirve para crear clases a partir de otras clases que ya existen. La nueva clase, va a heredar todos los atributos y las funciones que se hayan definido para la clase de la que hereda, modificándolos lo mínimo para adaptarse a la nueva realidad.

La clase de la que se hereda se llamará *"clase base"* y la clase que se construye heredando de ella, se llamará "*clase derivada*". Para señalar que una clase derivada hereda de una clase base, en su definición se establecerá la clase derivada y entre paréntesis la clase base.

Recordando apartados anteriores, teníamos la clase *"Punto"*. Definíamos la clase y sus funciones como:

```
class Punto(object):
"""Representa un punto sobre un plano con sus coordenadas cartesianas"""

    def __init__(self, x=0, y=0):
            self.x = float(x)
            self.y = float(y)

    def __str__(self):
        return "x = {}, y = {}".format(self.x, self.y)

    def __len__(self):
        return 0

    def __add__(self, otro_punto):
        return Punto(self.x + otro_punto.x, self.y + otro_punto.y)

    def __sub__(self, otro_punto):
        return Punto(self.x - otro_punto.x, self.y - otro_punto.y)

    def distancia(self, otro_punto):
        dx, dy = self.resta(otro_punto)
        dist = (dx**2 + dx**2)**0.5
        return dist

    def acimut(self, otro_punto):
        dx, dy = self.resta(otro_punto)
        acimut_rad = numpy.arctan2(dx, dy)
        acimut_gra = (acimut_rad * 200) / numpy.pi
        return (acimut_gra)

    def __resta(self, otro_punto):
        dx = self.x - otro_punto.x
        dy = self.y - otro_punto.y
        return (dx, dy)
```

Vamos a definir una nueva clase que herede de la clase *"Punto"*, será la clase *"Punto3D"*. Para definirla, en primer lugar, importaremos la clase *"Punto"* y a continuación definiremos la clase derivada heredando de la clase base.

```
from Punto import Punto

class Punto3D(Punto):
"""Representa un punto con sus coordenadas cartesianas y una cota"""
    def __init__(self, x, y, z):
        Punto.__init__(self, x, y)
        self.z = float(z)
```

En el código anterior, vemos como se define la herencia. Además, si nos fijamos en el constructor de la clase, una de las ventajas de usar la herencia es que podemos hacer una llamada directa al constructor de la clase base para definir los atributos que son comunes en la clase derivada. Veamos cómo funciona.

```
punto3d = Punto3D(3.5 , 5.1, 7)

print(punto3d.x)
print(punto3d.y)
print(punto3d.z)
```

```
3.5
5.1
7.0

Process finished with exit code 0
```

Otra ventaja de la herencia es que podremos utilizar todos los métodos que hemos definido para la clase base en la clase derivada. Por ejemplo, vamos a utilizar el método especial que *"__str__"* que definimos para la clase base:

```
punto3d = Punto3D(3.5 , 5.1, 7)

otro_punto = Punto3D(7.0, 12.1, 1.6)
print(punto3d.distancia(otro_punto))

print(punto3d)
```

```
4.949747468305833
x = 3.5, y = 5.1

Process finished with exit code 0
```

Sin embargo, aunque la clase derivada acepta los métodos de la clase base, solamente utilizar las propiedades que sean comunes a ambas. Esto será interesante en algunas funciones, como por ejemplo, en nuestro caso, la función que calcula el acimut, pero no lo será tanto en otras, como por ejemplo la función distancia, ya que realmente nos está devolviendo la distancia plana y no la distancia entre los dos puntos en tres dimensiones.

Vamos a ver como modificaríamos estas funciones en la clase derivada para que hiciese uso igualmente de las de la clase base, pero adaptándose a los atributos de la nueva clase.

```
from Punto import Punto

class Punto3D(Punto):
"""Representa un punto con sus coordenadas cartesianas y una cota"""
    def __init__(self, x, y, z):
        Punto.__init__(self, x, y)
        self.z = float(z)

    def __str__(self):
        return "{}, z = {}".format(Punto.__str__(self), self.z)

    def distancia(self, otro_punto):
        dx = self.x - otro_punto.x
        dy = self.y - otro_punto.y
        dz = self.z - otro_punto.z
        dist = (dx**2 + dy**2 + dz**2)**0.5
        return dist
```

Hemos definido el método especial "__str__" apoyándonos en el mismo método ya creado para la clase base, de modo que cuando llamemos al método desde una instancia de la clase derivada se usará este nuevo método en lugar del anterior, que solo se usará para la clase base. En la segunda función, hemos optado por una redefinición completa de la misma, obteniendo el mismo resultado.

```
punto3d = Punto3D(3.5 , 5.1, 7)
otro_punto = Punto3D(7.0, 12.1, 1.6)
print(punto3d.distancia(otro_punto))
print(punto3d)

p = Punto(8, 9)
q = Punto(1,2)
print(p.distancia(q))
print(p)
```

```
7.325298628724975
x = 3.5, y = 5.1, z = 7.0
9.899494936611665
x = 8.0, y = 9.0

Process finished with exit code 0
```

A la hora de utilizar la herencia, podremos derivar cuantas clases queramos e incluso derivar clases de clases derivadas. Cabe tener en cuenta, que siempre habrá que seguir una jerarquía entre clases, de modo que todas las clases que deriven de una clase base, sean realmente objetos del mundo real derivados de la misma. Por ejemplo, podemos tener una clase base *"persona"* y de ella derivar la clase *"trabajadores"* y la clase *"estudiantes"* e incluso de la clase trabajadores podríamos derivar subclases según la profesión.

Lo que no podríamos hacer es una clase base *"coche"* y derivar de ella una clase *"rueda"*, pues si bien en mundo real una rueda forma parte de un coche, no es un subconjunto del mismo, sino que es una propiedad.

Python nos ofrece la posibilidad de diseñar una clase derivada a partir de más de una clase base, es lo que se denomina "*Herencia múltiple"*, sin embargo, este tipo de herencia ya se escapa al alcance de este curso.

1.7 Ejercicios propuestos

A continuación, se propone una serie de ejemplos de ejercicios para aplicar los conceptos estudiados. Las soluciones a dichos ejercicios, se encuentran al final del documento, pero se recomienda al alumno que trate de desarrollarlos.

1.7.1 Ejercicio 1

Modela un objeto que defina una mesa. Las propiedades que debe tener son:

- Altura.
- Ancho.
- Largo.
- Número de patas.
- Color.

Debe tener una función que calcule el área de la mesa y otra que diga si es apta para mesa de trabajo en función de si su altura supera o no los 70 cm.

Define un script para crear un objeto de tipo mesa obteniendo sus propiedades por teclado y nos diga si es apta para trabajar, en cuyo caso nos dará la superficie útil.

```
Introduce el alto de la mesa (en metros): 0.65

Introduce el ancho de la mesa (en metros): 1

Introduce el largo de la mesa (en metros): 2

Introduce número de patas de la mesa: 4

Introduce el color de la mesa: Negro

La mesa no es apta para trabajo


Process finished with exit code 0


Introduce el alto de la mesa (en metros): 0.75

Introduce el ancho de la mesa (en metros): 1.2

Introduce el largo de la mesa (en metros): 2.5

Introduce número de patas de la mesa: 4

Introduce el color de la mesa: Negro

La mesa es apta para trabajo y tiene una superficie de 3.0 metros cuadrados


Process finished with exit code 0
```

1.7.2 Ejercicio 2

Modelar un objeto triángulo cuyas propiedades sean la longitud de dos de sus lados y el ángulo que forman entre ellos. Debe tener los siguientes métodos:

- Constructor.
- Conversor a texto.
- Calculadora de área.
- Calculadora de perímetro.
- Calculadora del tercer lado.

Modelar un segundo objeto llamado terreno. Su constructor ha de recibir una lista de triángulos. Debe tener los siguientes métodos:

- Constructor.
- Conversor a texto.
- Añadir triángulo (objeto triangulo).
- Añadir triangulo (dos lados y ángulo)
- Calcular área total.

Simulación de uso:

```
1  from Triangulo import Triangulo
2  from Terreno import Terreno
3
4  tri_1 = Triangulo(12, 12, 100)
5  tri_2 = Triangulo(15, 21, 91)
6  tri_3 = Triangulo(91, 65, 15)
7
8  print(tri_1)
9  print(tri_1.area())
10 print(tri_1.perimetro())
11 print(tri_1.tercerLado())
```

```
Lado 1: 12.0, lado 2: 12.0, ángulo interior: 100.0
72.0
40.970562748477136
16.97056274847714

Process finished with exit code 0
```

```
13 terreno = Terreno([tri_1, tri_2, tri_3])
14 print(terreno)
15 print(terreno.area())
```

```
Terreno:
    Triángulo 1: Lado 1: 12.0, lado 2: 12.0, ángulo interior: 100.0
    Triángulo 2: Lado 1: 15.0, lado 2: 21.0, ángulo interior: 91.0
    Triángulo 3: Lado 1: 91.0, lado 2: 65.0, ángulo interior: 15.0.
918.343389694198

Process finished with exit code 0
```

```
17 tri_4 = Triangulo(1, 9, 13)
18 terreno.addTriangulo(tri_4)
19 print(terreno)
20 print(terreno.area())
```

```
Terreno:
    Triángulo 1: Lado 1: 12.0, lado 2: 12.0, ángulo interior: 100.0
    Triángulo 2: Lado 1: 15.0, lado 2: 21.0, ángulo interior: 91.0
    Triángulo 3: Lado 1: 91.0, lado 2: 65.0, ángulo interior: 15.0
    Triángulo 4: Lado 1: 1.0, lado 2: 9.0, ángulo interior: 13.0.
919.2559325233022

Process finished with exit code 0
```

```
22 terreno.addTrianguloXMedidas(12, 13, 75)
23 print(terreno)
24 print(terreno.area())
```

```
Terreno:
    Triángulo 1: Lado 1: 12.0, lado 2: 12.0, ángulo interior: 100.0
    Triángulo 2: Lado 1: 15.0, lado 2: 21.0, ángulo interior: 91.0
    Triángulo 3: Lado 1: 91.0, lado 2: 65.0, ángulo interior: 15.0
    Triángulo 4: Lado 1: 1.0, lado 2: 9.0, ángulo interior: 13.0
    Triángulo 5: Lado 1: 12.0, lado 2: 13.0, ángulo interior: 75.0.
991.3185360591826

Process finished with exit code 0
```

1.7.3 Ejercicio 3

Escribe una clase que defina un circulo a partir de su radio. La clase tendrá un método que devuelva el área y otro que devuelva el perímetro. Desde un script se construirá un objeto de tipo circulo y se imprimirá el perímetro y el área.

```
Introduce el radio de la circunferencia: 5

El perímetro es 31.41592653589793

El radio es 78.53981633974483


Process finished with exit code 0
```

2 Soluciones a los ejercicios

2.1 Ejercicio 1

Modela un objeto que defina una mesa. Las propiedades que debe tener son:

- Altura.
- Ancho.
- Largo.
- Número de patas.
- Color.

Debe tener una función que calcule el área de la mesa y otra que diga si es apta para mesa de trabajo en función de si su altura supera o no los 70 cm.

Define un script para crear un objeto de tipo mesa obteniendo sus propiedades por teclado y nos diga si es apta para trabajar, en cuyo caso nos dará la superficie útil.

"*Mesa.py*"

```
1  class Mesa(object):
2
3      def __init__(self, alto, ancho, largo, patas, color):
4          self.alto = alto
5          self.ancho = ancho
6          self.largo = largo
7          self.patas = patas
8          self.color = color
9
10     def area(self):
11         return self.ancho * self.largo
12
13     def aptaTrabajo(self):
14         if self.alto >= 0.70:
15             return True
16         else:
17             return False
```

Para definir la clase hemos creado un archivo independiente. Hacemos lo siguiente:

- En la línea 1 definimos la clase.
- En la línea 3 definímos el constructor de la clase con las propiedades que se nos piden en el ejercicio.
- Entre las líneas 4 y 8 asignamos las propiedades. No hemos puesto ningún control para comprobar si los datos son numéricos o no. Si se hiciese en producción, sería recomendable hacerlo.
- En la línea 10 definimos la función que calcula el área de la mesa.
- En la línea 13 definimos la función que evalúa si es apta para trabajar.

```
1  from Mesa import Mesa
2
3  alto = float(input('Introduce el alto de la mesa (en metros): '))
4  ancho = float(input('Introduce el ancho de la mesa (en metros): '))
5  largo = float(input('Introduce el largo de la mesa (en metros): '))
6  patas = int(input('Introduce número de patas de la mesa: '))
7  color = input('Introduce el color de la mesa: ')
8
9  mesa = Mesa(alto, ancho, largo, patas, color)
10
11 if mesa.aptaTrabajo():
12     print('La mesa es apta para trabajo y tiene una superficie de', mesa.area(),
13           'metros cuadrados')
14 else:
15     print('La mesa no es apta para trabajo')
```

```
Introduce el alto de la mesa (en metros): 0.65
Introduce el ancho de la mesa (en metros): 1
Introduce el largo de la mesa (en metros): 2
Introduce número de patas de la mesa: 4
Introduce el color de la mesa: Negro
La mesa no es apta para trabajo

Process finished with exit code 0

Introduce el alto de la mesa (en metros): 0.75
Introduce el ancho de la mesa (en metros): 1.2
Introduce el largo de la mesa (en metros): 2.5
Introduce número de patas de la mesa: 4
Introduce el color de la mesa: Negro
La mesa es apta para trabajo y tiene una superficie de 3.0 metros cuadrados

Process finished with exit code 0
```

En otro archivo hemos creado el programa principal que hace lo siguiente:

- En la línea 1 importa el módulo donde está definida la clase “Mesa”.
- Entre las líneas 3 y 7 pide al usuario las distintas propiedades del objeto.
- En la línea 9 crea una instancia de la clase “Mesa”.
- Entre las líneas 11 se pregunta si es apta para trabajo haciendo uso de la función específica.

2.2 Ejercicio 2

Modelar un objeto triángulo cuyas propiedades sean la longitud de dos de sus lados y el ángulo que forman entre ellos. Debe tener los siguientes métodos:

- Constructor.
- Conversor a texto.
- Calculadora de área.
- Calculadora de perímetro.
- Calculadora del tercer lado.

Modelar un segundo objeto llamado terreno. Su constructor ha de recibir una lista de triángulos. Debe tener los siguientes métodos:

- Constructor.
- Conversor a texto.
- Añadir triángulo (objeto triangulo).
- Añadir triangulo (dos lados y ángulo)
- Calcular área total.

"Triangulo.py"

```
1  import numpy
2
3  class Triangulo(object):
4
5      def __init__(self, lado1, lado2, angulo):
6          self.lado1 = float(lado1)
7          self.lado2 = float(lado2)
8          self.angulo = float(angulo)
9
10     def __str__(self):
11         return 'Lado 1: {}, lado 2: {}, ángulo interior: {}'.format(self.lado1,
12                                                                     self.lado2,
self.angulo)
13
14     def area(self):
15         angulo_rad = self.angulo * numpy.pi / 200
16         area = (self.lado1 * self.lado2 * numpy.sin(angulo_rad))/2
17         return area
18
19     def tercerLado(self):
20         angulo_rad = self.angulo * numpy.pi / 200
21         lado3 = ((self.lado1 ** 2 + self.lado2 ** 2) - (2 * self.lado1 *
self.lado2
22                                                         * numpy.cos(angulo_rad))) **
0.5
23         return lado3
24
25     def perimetro(self):
26         return self.lado1 + self.lado2 + self.tercerLado()
```

Definimos el módulo "Triangulo.py" en el que vamos a crear la clase Triangulo. Lo analizamos:

- En la línea 1 importamos el módulo numpy que necesitaremos para hacer operaciones matemáticas.
- En la línea 3 definimos la clase Triangulo.
- En la línea 5 creamos el constructor de clase y en las siguientes asignamos propiedades.
- En la línea 10 definimos la función que formateará a texto las instancias de la clase triangulo.
- En la línea 14 definimos la función que calcula el área del triángulo.
- En la línea 19 la función que calcula el tercer lado del triángulo.
- En la línea 25 la función que calcula el perímetro.

"Terreno.py"

```
1  from Triangulo import Triangulo
2
3  class Terreno(object):
4
5      def __init__(self, triangulos):
6          if isinstance(triangulos, list):
7              for el in triangulos:
8                  if not isinstance(el, Triangulo):
9                      raise Exception('Los elementos de la lista no son
   triángulos')
10             self.triangulos = triangulos
11         else:
12             raise Exception('Se ha de introducir una lista')
13
14     def __str__(self):
15         texto = 'Terreno:'
16         for i in range(0, len(self.triangulos)):
17             texto += '\n    Triángulo {}: {}'.format(i+1,
   str(self.triangulos[i]))
18         return texto
19
20     def addTriangulo(self, triangulo):
21         if isinstance(triangulo, Triangulo):
22             self.triangulos.append(triangulo)
23         else:
24             raise Exception('El elemento que se intenta añadir no es un
   triángulo')
25
26     def addTrianguloXMedidas(self, lado1, lado2, angulo):
27         triangulo = Triangulo(lado1, lado2, angulo)
28         self.triangulos.append(triangulo)
29
30     def area(self):
31         area = 0.0
32         for triangulo in self.triangulos:
33             area += triangulo.area()
34         return area
```

Posteriormente creamos el módulo "Terreno.py" que consta de las siguientes partes:

- En la línea 1 importa el módulo "Triángulo", ya que lo necesitaremos para definir las unidades básicas del terreno.
- En la línea 3 definimos la clase Terreno.
- En la línea 5 definimos el constructor de la clase y en las siguientes líneas asignamos propiedades comprobando que los elementos con los que intentamos construir el Terreno sean instancias de Triangulo.
- En la línea 14 definimos la salida como texto.
- En la línea 20 definimos una función que nos permita añadir triángulos a nuestro terreno.
- En la línea 26 definimos otra función que nos permita añadir a nuestro terreno un triángulo

a partir de sus datos en lugar de tener que generar primero la instancia de triángulo.

- Finalmente, en la línea 30, creamos una función que devuelve el área, calculándola como la suma de las áreas de todos los triángulos que forman el terreno.

Simulación de uso:

```
1  from Triangulo import Triangulo
2  from Terreno import Terreno
3
4  tri_1 = Triangulo(12, 12, 100)
5  tri_2 = Triangulo(15, 21, 91)
6  tri_3 = Triangulo(91, 65, 15)
7
8  print(tri_1)
9  print(tri_1.area())
10 print(tri_1.perimetro())
11 print(tri_1.tercerLado())
```

```
Lado 1: 12.0, lado 2: 12.0, ángulo interior: 100.0
72.0
40.970562748477136
16.97056274847714

Process finished with exit code 0
```

```
13 terreno = Terreno([tri_1, tri_2, tri_3])
14 print(terreno)
15 print(terreno.area())
```

```
Terreno:
    Triángulo 1: Lado 1: 12.0, lado 2: 12.0, ángulo interior: 100.0
    Triángulo 2: Lado 1: 15.0, lado 2: 21.0, ángulo interior: 91.0
    Triángulo 3: Lado 1: 91.0, lado 2: 65.0, ángulo interior: 15.0
918.343389694198

Process finished with exit code 0
```

```
17 tri_4 = Triangulo(1, 9, 13)
18 terreno.addTriangulo(tri_4)
19 print(terreno)
20 print(terreno.area())
```

```
Terreno:
    Triángulo 1: Lado 1: 12.0, lado 2: 12.0, ángulo interior: 100.0
    Triángulo 2: Lado 1: 15.0, lado 2: 21.0, ángulo interior: 91.0
    Triángulo 3: Lado 1: 91.0, lado 2: 65.0, ángulo interior: 15.0
    Triángulo 4: Lado 1: 1.0, lado 2: 9.0, ángulo interior: 13.0
919.2559325233022

Process finished with exit code 0
```

```
terreno.addTrianguloXMedidas(12, 13, 75)
print(terreno)
print(terreno.area())
```

```
Terreno:
    Triángulo 1: Lado 1: 12.0, lado 2: 12.0, ángulo interior: 100.0
    Triángulo 2: Lado 1: 15.0, lado 2: 21.0, ángulo interior: 91.0
    Triángulo 3: Lado 1: 91.0, lado 2: 65.0, ángulo interior: 15.0
    Triángulo 4: Lado 1: 1.0, lado 2: 9.0, ángulo interior: 13.0
    Triángulo 5: Lado 1: 12.0, lado 2: 13.0, ángulo interior: 75.0
991.3185360591826

Process finished with exit code 0
```

2.3 Ejercicio 3

Escribe una clase que defina un circulo a partir de su radio. La clase tendrá un método que devuelva el área y otro que devuelva el perímetro. Desde un script se construirá un objeto de tipo circulo y se imprimirá el perímetro y el área.

"Circulo.py"

```
import numpy

class Circulo(object):

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

    def perimetro(self):
        return (2 * numpy.pi * self.radio)

    def area(self):
        return  (self.radio **2 * numpy.pi)
```

```
from Circulo import Circulo

radio = float(input('Introduce el radio de la circunferencia: '))

cir = circulo(radio)

print('El perímetro es', cir.perimetro())
print('El radio es', cir.area())
```

```
Introduce el radio de la circunferencia: 5
El perímetro es 31.41592653589793
El radio es 78.53981633974483

Process finished with exit code 0
```

3 Bibliografía

Bahit, E. (s.f.). *Python para principiantes*. Obtenido de https://librosweb.es/libro/python/

Foundation, P. S. (2013). *Python.org*. Obtenido de https://www.python.org/

Suárez Lamadrid, A., & Suárez Jiménez, A. (Marzo de 2017). *Python para impacientes*. Obtenido de http://python-para-impacientes.blogspot.com.es/2017/03/el-modulo-time.html