Clases y Objetos

En programación, un *objeto* es una estructura de datos que encapsula *datos*, ¡cómo no!, y las *operaciones* necesarias para trabajar con dicho objeto.

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

- Coche:
  - Estado:
    - posición
    - orientación
    - velocidad
    - etc.
  - Operaciones:
    - encender el motor o apagarlo
    - acelerar o frenar
    - girar
    - saber la velocidad
    - etc.

En el mundo de la programación orientada a objetos, el conjunto de los datos de un objeto se llama su *estado*, y cada uno de dichos datos es un *atributo*. A las operaciones se les llama *métodos*.

Otro ejemplo: la información sobre una *televisión* incluye su *estado* (esto es, sus datos: 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.)

Una cosa es la definición abstracta de una *clase* de objetos y otra cosa es *cada objeto* concreto de esa clase. A la descripción de un punto del plano genérico (por ejemplo, diciendo que los atributos serán sus dos coordenadas reales y que las operaciones serán unas u otras) se le llama *clase*. A cada punto concreto de dicha clase (por ejemplo, el de coordenadas (4.0, 5.0)) se le llama *objeto*.

Vamos a detallar esta situación en Python.

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`.

El método `__init__` se activa automáticamente cuando creamos un objeto de la clase `Point`. En nuestra definición, cada vez que creamos un punto nuevo, empieza teniendo sus dos coordenadas a cero.

Para referirnos a los atributos, dentro de la clase, usamos el identificador `self`. El parámetro `self` del método `__init__` se refiere al propio objeto `Point` que se está definiendo.

Cuando se crea un punto, se activa este método de inicialización y, en él, se crean los atributos `x` e `y`, inicialmente a cero.

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 entre dos puntos, esto esm entre dos objetos de la clase `Point`:

In [4]:
```
from math import sqrt, pi

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

Pongamos las anotaciones de tipo en eta función para mayor claridad:

In [5]:
```
def distancia(p0: Point, p1: Point) -> float:
    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. Con anotaciones de tipo:

In [6]:
```
def es_rectangulo(a: Point, b: Point, c: Point, d: Point) -> bool:
    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 = 3, 1
p2.x, p2.y = 0, 1
p3.x, p3.y = 3, 0

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

Out[6]: `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 inicial:

In [7]:
```
class Point(object):
    """
    Point class. It represents 2D points.

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

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

In [8]:
```
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 [9]:
```
print(p0)
p0
```

```
<__main__.Point object at 0x000002B52A85FAD0>
```

Out[9]:
```
<__main__.Point at 0x2b52a85fad0>
```

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

In [10]:
```
class Point(object):
    """
    clase Point. It represents 2D points.

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

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

    def __str__(self):
        """
        This method returns the string that represents a 2D point
        """
        return '({0:.2f}, {1:.2f})'.format(self.x, self.y)
```

In [11]:
```
p0 = Point(3.0, 4.0)
print(p0)
p0
```

```
(3.00, 4.00)
```

Out[11]:
```
<__main__.Point at 0x2b52a860d50>
```

Más métodos

Antes, definimos una función (*externa* a la clase `Point`) para calcular la distancia entre dos puntos. También podemos *encapsular* la función distancia dentro de la clase `Point`:

In [12]:
```
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 [13]:
```
p = Point(5, 6)
q = Point(6, 7)
print(p, q, p.distance(q))
```

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

In [14]:
```
def is_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)
is_rectangle(p0, p1, p2, p3)
```

Out[14]:
```
True
```

El método `distance` es meramente *de acceso*, pues su uso *no* modifica el objeto, sino que únicamente consulta información sobre su estado. Pero también hay métodos que *modifican* el estado del objeto. Pensemos, por ejemplo, en mover el objeto, aplicando una traslación:

In [15]:
```
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 [16]:
```
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 la clase `Vector`, la operación `move` puede diseñarse para que traslade un punto *según un vector*, más cómodamente, en lugar de tomar como parámetros las dos coordenadas de la traslación.

In [17]:
```
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 [18]:
```
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 [19]:
```
class Circle(object):
    """
        Class to represents a circle.
    """
    def __init__(self, center, radius):
        """
        Constructor

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

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

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

In [20]:
```
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 `circle`:

In [21]:
```
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 [22]:
```
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 [23]:
```
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

Para sumar vectores, podemos definir una operación `suma` :

```
add(u, v)
```

Pero querríamos también poder sumarlos con la operación `+` , y usarla con la sintaxis clásica como operación aritmética:

```
u + v
```

Algunas de estas operaciones (`<` , `<=` , `==` , `!=` , `>` , `>=`) pueden definirse en Python con nombres especiales:

```
object.__lt__(self, other)
object.__le__(self, other)
object.__eq__(self, other)
object.__ne__(self, other)
object.__gt__(self, other)
object.__ge__(self, other)
```

También hay otros métodos con fines especiales, tales como facilitar la conversión de un objeto en otro tipo de datos mediante la función de casting típica (ej.: `bool`)

```
object.__bool__(self, other)
```

La lista completa de métodos especiales puede encontrarse, por ejemplo, en la siguiente url:

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

In [24]:

```
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 represents a 2D vector based upon its cartesian coordi
```

```
    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 [25]:
```
p = Point(0,1)
v = Vector(2,2)
q = p + v
q.distance(p)
```

Out[25]:
```
2.8284271247461903
```

Ejemplo adicional

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

In [26]:

```
class Persona(object):
    """
    Esta clase representa la ficha 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: str
    edad: int
    estatura: float
    direccion: str
    """

    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, "Pozuelo de Alarcón")
print(p)
p
```

```
<Nombre: Blacky, Edad: 12, Estatura: 0.3...>
```

Out[26]:

```
<__main__.Persona at 0x2b52a88a290>
```

In [27]:

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

mi_agenda = dict()
mi_agenda["7023"] = p
mi_agenda["3401"] = Persona("Fer", 25, 1.85, "Pozuelo de Alarcón")
mi_agenda["5003"] = Persona("Artu", 28, 1.78, "Pozuelo de Alarcón")
mi_agenda["3045"] = Persona("Rosa", 89, 1.65, "Madrid")

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

mostrar_agenda(mi_agenda)
```

```
7023 <Nombre: Blacky, Edad: 12, Estatura: 0.3...>
3401 <Nombre: Fer, Edad: 25, Estatura: 1.85...>
5003 <Nombre: Artu, Edad: 28, Estatura: 1.78...>
3045 <Nombre: Rosa, Edad: 89, Estatura: 1.65...>
```

In [28]:
```
# 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: 26, Estatura: 1.9...>
324098 <Nombre: Elena, Edad: 60, Estatura: 1.7...>
534001 <Nombre: Fer, Edad: 25, Estatura: 1.85...>
450303 <Nombre: Artu, Edad: 30, Estatura: 1.78...>
376045 <Nombre: Rosa, Edad: 89, Estatura: 1.65...>
Pozuelo de Alarcón
```

Herencia y especialización

Imagina que deseamos definir una clase para los clientes de un supermercado. Cada `Cliente` es una `Persona` de la cual queremos registrar información adicional: su `cuenta` y el `importe_maximo` fijado para cada compra.

No necesitamos volver a definir toda la información como `Persona`. Podemos definir una clase nueva, `Cliente`, derivada a partir de la primera, `Persona`:

In [29]:
```
class Cliente(Persona):

    def __init__(self, nombre, edad, estatura, direccion, num_cta, imp_m
        Persona.__init__(self, nombre, edad, estatura, direccion)
        self.num_cuenta = num_cta
        self.importe_maximo = imp_max

    def __str__(self):
        return '<Nombre: ' + self.nombre + ', Edad: ' + str(self.edad) \
            + ', Núm. cuenta: ' + str(self.num_cuenta) \
            + ', Importe máx.: ' + str(self.importe_maximo) + '>'

    def es_cliente_preferente(self):
        return self.importe_maximo >= 500
```

In [30]:
```
# He aquí unas pocas pruebas:

cli_1 = Cliente("Javier", 57, 1.70, "Carretera de Húmera, Pozuelo de Ala

print()

cli_2 = Cliente("Elena", 62, 1.50, "Príncipe de Vergara, Madrid", "32549

print(cli_1)
print(type(cli_1))
print(cli_1.es_cliente_preferente())

print()

print(cli_2)
print(cli_2.es_cliente_preferente())
```

```
<Nombre: Javier, Edad: 57, Núm. cuenta: 002445281325490654, Importe má
x.: 300>
<class '__main__.Cliente'>
False

<Nombre: Elena, Edad: 62, Núm. cuenta: 325490654002445281, Importe má
x.: 600>
True
```

La clase `Persona` es la *superclase*. La clase `Cliente` es la subclase, es decir, la clase *derivada*, que ha heredado la estructura de la superclase, es decir, sus atributos y métodos, añadiendo quizás algun atributo (como `num_cuenta` y `importe_maximo`), redefiniendo algunos de ellos (como `__str__`) o añadiendo otros nuevos (como `es_cliente_preferente`).

Apéndice: una clase abstracta e implementaciones

Es habitual implementar números complejos mediante sus componentes cartesianas, *parte real* y *parte imaginaria*. Pero también lo es mediante su representación en forma polar, esto es, con su *módulo* y *argumento*. Una clase abstracta define la signatura de un objeto, esto es, sus operaciones y argumentos. Es como sigue; más abajo veremos cuál es la finalidad de esto.

In [31]:

```
class Complex(object):
    """
    This class represents a complex number

    Rest of docstring to be fulfilled
    """
    def __init__(self, arg1, arg2, mode="cartesian"):
        """
        docstring to be fulfilled

        If mode="cartesian", arg1 and arg2 represent real and imag compo
        If mode="polar", arg1 and arg2 represent module and argument com
        Other mode will raise an exception
        """
        raise Exception("Class Vector has no implemented the __init__ me

    def __str__(self, mode="polar"):
        """
        This method returns str representing the cartesian or polar form
        """
        raise Exception("Class Complex has no implemented the __str__ me

    def module(self):
        """
        docstring to be fulfilled
        """
        raise Exception("Class Complex has no implemented the module met

    def __add__(self, v):
        """
        docstring to be fulfilled
        """
        raise Exception("Class Complex has no implemented the __add__ me
```

Como hemos visto, en una clase abstracta no se concreta la representación interna de la clase. Sirve para fijar ideas de lo que se quiere definir y su comportamiento previsto, pero sin concretar la representación. Esto se hará cuando se diseñe por completo, tras tomar una decisión sobre la representación interna de dicho objeto. Por ejemplo, así:

In [32]:

```
import math

class PolarComplex(Complex):
    """
    This class implements complex number, based upon its polar represent

    Attributes:
    -----------
    __module, __argument: real
    """
    def __init__(self, arg1, arg2, mode="cartesian"):
        """
        docstring to be fulfilled
        """
        if mode=="cartesian":
            self.__module = math.sqrt(px**2 + py**2)
            self.__argument = math.arctan(py/px)
        elif mode=="polar":
            self.__module = px
            self.__argument = py

    def __str__(self, mode="polar"):
        """
        This method returns str representing the cartesian or polar form
        """
        raise Exception("Class PolarVector has no implemented the __str_

    def module(self):
        """
        docstring to be fulfilled
        """
        raise Exception("Class PolarVector has no implemented the module

    def __add__(self, v):
        """
        docstring to be fulfilled
        """
        raise Exception("Class PolarVector has no implemented the __add_
```

Ejercicios

1. Completa esta clase y documéntala. Diseña un pequeño conjunto de casos de prueba para ver el comportamiento de estas operaciones.
2. Añade una operación para calcular el conjugado de un número complejo.
3. Añade una operación para calcular el producto de un complejo por un escalar (float) y el producto de dos complejos. ¿Podrían unificarse estas dos operaciones en un solo método? Este ejercicio presenta nu ejemplo muy claro de *polimorfismo*.
4. Diseña otra clase para trabajar con complejos, con la misma *signatura* que la anterior, pero basada, en esta ocasión, en una representación cartesiana. Ambas implementaciones deben comportarse de idéntica manera. Compruébalo con los mismos casos de prueba que la anterior.

Implementaciones de una clase abstracta

Las dos clases anteriores, `PolarComplex` y `CartesianComplex` deberían comportarse de manera idéntica. No obstante, se basan en *mecanismos* distintos, es decir, representaciones internas distintas. Y las operaciones en cada una de ellas actúan manejando dichas representaciones distintas.

La abstracción es fundamental en programación: en el mundo de los tipos abstractos de datos, consiste en separar lo que se ve (el comportamiento) e ignorar los detalles internos (la implementación). Esta idea es semejante a conducir dos coches, uno de gasolina y otro eléctrico, pero ambos con volantes iguales, mismos mandos de arranque, mismo acelerador, mismo freno, encendido de luces, etc.

Cuando consideramos dos implementaciones distintas para un mismo concepto, es posible que una de las representaciones sea ventajosa (más eficiente) para algunas operaciones en detrimento quizá de otras, más ineficientes, mientras que, en la segunda representación, haya otras operaciones más eficientes y sea más adecuada en otras aplicaciones. Estas diferencias serán los argumentos para implementar y preferir una de las representaciones frente a la otra, según el uso que se vaya a hacer explotando preferentemente unas operaciones u otras.

Esta idea básica se ha de tener en cuenta a la hora de estudiar el catálogo clásico de estructuras de datos y a la hora de diseñar una clase nueva para nuestros fines.

In [33]:
```
class MyClass():
    def __init__(self):
        self.x = 1
        self._y = 2
        self.__z = 3

my_ob = MyClass()
```

Apéndice: nombres de atributos: guiones bajos

Partimos de un ejemplo para facilitar la explicación técnica. Luego vendrá la explicación conceptual. Observa los nombres de los atributosde la siguiente clase y en su inicio: los hay con un guion bajo, con dos o con ninguno.

In [34]:
```
print(my_ob.x)
print(my_ob._y)
```

```
1
2
```

Hasta aquí, todo va bien. Los atributos `x` e `_y` se están manejando desde fuera de la clase.

Podemos verlos o modificarlos sin problema. La diferencia entre nombrarlos sin o con guion bajo es únicamente una convención, para señalar que se trata de atributos de la clase, y es una buena costumbre, aunque no hemos entrado en este detalle hasta ahora para evitar esta disquisición en un primer contacto con el mundo de los objetos.

Identificar un atributo empezando con un guion bajo indica al usuario que se trata de un atributo de la clase, aunque en realidad el usuario puede usarlo, o sea, fuerza a que el usuario sea más consciente de que está empleando un atributo que en realidad es del ámbito interno, local, de la clase.

Pero veamos qué ocurre con el otro atributo:

In [35]:
```
print(my_ob.__z)
```

```
---------------------------------------------------------------------
----
AttributeError                            Traceback (most recent call l
ast)
Cell In[35], line 1
----> 1 print(my_ob.__z)

AttributeError: 'MyClass' object has no attribute '__z'
```

Este atributo no puede manejarse desde fuera de la clase sí puede de la manera convencional, con su identificador, aunque en realidad sí puede ser accesible de otro modo, un poco más retorcido:

In [36]:
```
print(my_ob._MyClass__z)
```

```
3
```

Es una advertencia aún más fuerte, imponiendo una manera de acceder al atributo local mucho más incómoda, incluso algo absurda, imponiendo una dificultad adicional, que realmente muy pocos programadores usan, como debe ser, y que está limitada a pruebas en el curso del desarrollo, más que en el uso convencional de los atributos de una clase.

Más técnicamente, es una indicación de la distinción entre atributos privados (_x) y protegidos (__y). Pero únicamente es eso: una indicación.

La explicación conceptual

En principio, los atributos de una clase son características propias suyas, internas. Usarlas desde fuera es una irregularidad, restringida normalmente al usuario convencional de una clase, que debería usar los atributos del objeto, normalmente, a través de las operaciones de acceso y modificación.

En muchos lenguajes de programación, estos atributos son privados realmente, y no es posible acceder a ellos, más que a través de las operaciones de acceso y modificación de la clase. En Python, el acceso a dichos atributos privados es accesible, sobre todo, para facilitar el desarrollo y las pruebas durante el desarrollo de una clase. Pero sin perder de vista lo que son atributos privados, en realidad *seudoprivados*, y los dos niveles de acceso a los mismos.

Bibliografía

- M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Python*, Ed. Wiley