Características de la POO en *Python* (resumen)

Programación Orientada a Objetos

La programación orientada a objetos, o POO para abreviar, es un paradigma de programación que proporciona un medio de estructurar programas para que las propiedades y los comportamientos se agrupen en objetos individuales.

Por ejemplo, un objeto podría representar a una persona con un nombre de propiedad, edad, dirección, etc., con comportamientos como caminar, hablar, respirar y correr. O un correo electrónico con propiedades como lista de destinatarios, asunto, cuerpo, etc., y comportamientos como agregar archivos adjuntos y enviar.

Dicho de otra manera, la programación orientada a objetos es un enfoque para modelar cosas concretas del mundo real, como los automóviles, así como las relaciones entre cosas como compañías y empleados, estudiantes y maestros, etc. POO modela entidades del mundo real como objetos de software, que tienen algunos datos asociados con ellos y pueden realizar ciertas funciones.

> NOTA: Dado que Python es un lenguaje de programación

> multiparadigma, se puede elegir el paradigma que mejor se adapte al problema en cuestión, mezclar diferentes paradigmas en un programa y/o cambiar de un paradigma a otro a medida que un programa evoluciona.

Al igual que muchos lenguajes, Python permite definir clases que encapsulan datos y las funciones que operan en ellos. Los usaremos a veces para hacer que nuestro código sea más limpio y simple. Está basada en varias técnicas, como las siguientes:

- herencia
- cohesión
- abstracción
- polimorfismo
- acoplamiento
- encapsulación

Algunas particularidades de POO en Python son las siguientes:

- Todo es un objeto, incluyendo los tipos y clases.
- Permite herencia múltiple.
- No existen métodos ni atributos privados.
- Los atributos pueden ser modificados directamente.
- Permite “monkey patching” (reemplazo dinámico de métodos/atributos en tiempo de ejecución).
- Permite “duck typing” (tipificación dinámica).
- Permite la sobrecarga de operadores.
- Permite la creación de nuevos tipos de datos.

A continuación se procede a definir algunos conceptos necesarios para entender la POO en Python:

Clases y Objetos

Las clases son los modelos sobre los cuáles se construirán nuestros objetos. Las clases definen objetos. Son un plano para crear objetos, definen los datos (atributos) y la funcionalidad (métodos) de los objetos. Las clases se utilizan para crear nuevas estructuras de datos definidas por el usuario que contienen información arbitraria sobre algo y las funcionalidades para actuar sobre esas estructuras de datos.

Los objetos son una pieza de datos encapsulados con funcionalidades en un programa Python que se construye de acuerdo con una definición de clase. A menudo, un objeto corresponde a una cosa en el mundo real. Un objeto consiste en un número arbitrario de atributos y métodos, encapsulados dentro de una sola unidad. Resumiendo, un objeto es una colección de datos con comportamientos

asociados.

Los objetos son **instancias** de clases que se pueden asociar entre sí. Una instancia de una clase es un objeto específico con su propio conjunto de datos y comportamientos. El proceso de crear un objeto a partir de una clase se denomina instanciación.

El propósito clave de modelar un objeto en un diseño orientado a objetos es determinar cuál será la interfaz pública de ese objeto. La interfaz es la colección de atributos y métodos que otros objetos pueden usar para interactuar con ese objeto. No necesitan, y a menudo no se les permite, acceder al funcionamiento interno del objeto.

Un ejemplo común del mundo real es la televisión. Nuestra interfaz con la televisión es el control remoto. Cada botón del control remoto representa un método que se puede invocar en el objeto de televisión. Cuando nosotros, como objeto de llamada, accedemos a estos métodos, no sabemos ni nos importa si el televisor recibe su señal de una antena, una conexión de cable o una antena parabólica. No nos importa qué señales electrónicas se envían para ajustar el volumen, o si la imagen está preprocesada.

Este proceso de ocultar la implementación, o detalles funcionales, de un objeto se denomina **ocultación de información**. A veces también se conoce como **encapsulación**, pero la encapsulación es en realidad un término que lo abarca todo. La distinción entre encapsulación y ocultación de información es en gran medida irrelevante, especialmente a nivel de diseño. Muchas referencias prácticas usan estos términos indistintamente. Como programadores de Python, en realidad no tenemos o necesitamos ocultar información verdadera, por lo que la definición más amplia para la encapsulación es adecuada.

La interfaz pública, sin embargo, es muy importante. Necesita ser cuidadosamente diseñada ya que es difícil cambiarla en el futuro. Cambiar la interfaz interrumpirá los objetos del cliente que la estén llamando. Podemos cambiar las partes internas todo lo que queramos, por ejemplo, para hacerlo más eficiente, o para acceder a los datos a través de la red y localmente, y los objetos del cliente aún podrán hablar con él, sin modificaciones, utilizando el público interfaz. Pero si cambiamos la interfaz cambiando los nombres de los atributos a los que se accede públicamente, o alterando el orden o los tipos de argumentos que un método puede aceptar, todos los objetos del cliente también tendrán que modificarse.

La **abstracción** es otro concepto orientado a objetos relacionado con la encapsulación y la ocultación de información. En pocas palabras, la abstracción significa tratar con el nivel de detalle más apropiado para una tarea determinada. Es el proceso de extraer una interfaz pública de los detalles internos. El conductor de un automóvil necesita interactuar con la dirección, el acelerador y

los frenos. El funcionamiento del motor, el tren de transmisión y el subsistema de frenos no le importan al conductor. Un mecánico, por otro lado, trabaja en un nivel diferente de abstracción, ajusta el motor y desangra los frenos. Este es un ejemplo de dos niveles de abstracción para un automóvil.

Crear clases de Python

La clase más simple en Python 3 se define así:

```
class Punto:
    pass
```

La definición de clase comienza con la palabra clave `class` . Esto es seguido por un nombre (de nuestra elección) que identifica la clase, y termina con dos puntos. La línea de definición de clase es seguida por el contenido de la clase identado.

> **PEP 8: Clases**. El nombre de las clases se define en singular, utilizando notación CamelCase (comienza con una letra mayúscula y cualquier palabra posterior también debe comenzar con una mayúscula).

Atributos/Miembros/Propiedades

Los atributos son las características intrínsecas del objeto. Una variable definida para una clase (atributo de clase) o para un objeto (atributo de instancia) permite empaquetar datos en unidades cerradas (clases o instancias). Los datos generalmente representan las características individuales de un determinado objeto. Una clase puede definir conjuntos específicos de características que comparten todos los objetos de esa clase.

Los atributos se denominan frecuentemente *miembros* o *propiedades*. Algunos autores sugieren que los términos tienen significados diferentes, generalmente que los miembros son configurables, mientras que las propiedades son de solo lectura. En Python, el concepto de "solo lectura" no se existe, todas las propiedades son públicas.

Para añadir atributos no tenemos que hacer nada especial en la definición de clase. Podemos establecer atributos arbitrarios en un objeto instanciado usando la notación punto:

```
<objeto>.<atributo> = <valor>
```

Con esta notación asignamos un valor a un atributo. El valor puede ser cualquier cosa: una primitiva Python, un tipo de datos incorporado u otro objeto. Incluso

In [1]:
```python
class Punto:
    pass

# instanciamos objetos a partir de la clase
p1 = Punto()
p2 = Punto()

# accedemos a los atributos de la instacia (objeto)
# "monkey patching"
p1.x = 5
p1.y = 4
p2.x = 3
p2.y = 6

print(p1.x, p1.y)
print(p2.x, p2.y)
```

```
5 4
3 6
```

Aunque se puede utilizar la notación punto para definir dinámicamente un atributo (se define durante la ejecución del programa), lo habitual es definir los atributos como variables dentro del cuerpo de la clase. Dependiendo de su naturaleza, estos podrán ser:

- Atributos de clase (*o variable de clase, variable estática, atributo estático*): una variable que se crea estáticamente en la definición de clase y que todos los objetos de clase comparten. Se definen fuera de cualquier método.
- Atributos de instancia (*o variable de instancia*): una variable que contiene datos que pertenecen solo a una sola instancia. Otras instancias no comparten esta variable (en contraste con los atributos de clase). Se definen dentro de un método (normalmente el método `__init__` encargado de la inicialización de la clase) y se referencian con la palabra clave `self` que hace referencia al propio objeto.

> **PEP 8: propiedades**. Las propiedades se definen de la misma forma que las variables (aplican las mismas reglas de estilo).

In [2]:
```python
class Punto:
    # variables de clase compartida por todas las instancias
    origen_x = 0
    origen_y = 0

    def __init__(self, x, y):
        # variables de instancia única para cada instancia
        self.x = x
        self.y = y

p1 = Punto(1, 2)
p2 = Punto(3, 4)
print(p1.origen_x, p1.origen_y, p1.x, p1.y)
print(p2.origen_x, p2.origen_y, p2.x, p2.y)
```

```
0 0 1 2
```

```
0 0 3 4
```

Métodos

Los métodos definen los comportamientos que se pueden realizar en una clase específica de objetos. En el nivel de programación, los métodos son como funciones en la programación estructurada, pero tienen acceso a todos los datos asociados del objeto en el que se definene. Al igual que las funciones, los métodos también pueden aceptar parámetros y devolver valores. Son por tanto, un subconjunto de la funcionalidad general de un objeto.

El método se define de manera similar a una función (usando la palabra clave **`def`**) en la definición de clase. Un objeto puede tener un número arbitrario de métodos.

> **NOTA**: Los nombres de atributos anulan los nombres de método con el mismo valor. Para evitar conflictos de nombres accidentales, que pueden causar errores difíciles de encontrar en programas grandes, es aconsejable utilizar algún tipo de convención que minimice la posibilidad de conflictos. Las convenciones posibles incluyen el uso de verbos para métodos y sustantivos para los atributos de datos.

In [3]:

```
class Punto:
    origen_x = 0
    origen_y = 0

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

    def mover_a(self, x, y):
        self.x = x
        self.y = y

p = Punto(0, 0)
print(p.x, p.y)
p.mover_a(1, 2)
print(p.x, p.y)
```

```
0 0
1 2
```

La única diferencia entre los métodos y las funciones normales es que todos los métodos tienen un argumento requerido. Este argumento se denomina convencionalmente **`self`** y es simplemente una referencia al objeto sobre el que se invoca el método. Podemos acceder a los atributos y métodos de ese objeto como si fuera cualquier otro objeto. Esto es exactamente lo que hacemos dentro del método `mover_a` cuando establecemos los atributos `x` e `y` del objeto propio.

Cuando llamamos a un método, no tenemos que pasarle el argumento propio. Python se encarga automáticamente de esto por nosotros. Sabe que estamos llamando a un método de un objeto, por lo que automáticamente pasa ese objeto al método.

Un método realmente es solo una función que está en una clase. En lugar de invocar el método en el objeto, podemos invocar la función en la clase, pasando explícitamente nuestro objeto como argumento propio:

In [4]:
```
# La salida es la misma que en el ejemplo anterior porque internamen
p = Punto(0, 0)
Punto.mover_a(p, 1, 2)
# equivalente a p.mover_a(1, 2)
print(p.x, p.y)
```

```
1 2
```

In [5]:
```
import math

class Punto:
    origen_x = 0
    origen_y = 0

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

    def mover_a(self, x, y):
        self.x = x
        self.y = y

    def distancia_a(self, otro_punto):
        return math.sqrt((self.x - otro_punto.x)**2 + (self.y - otro

p1 = Punto(0, 0)
p2 = Punto(0, 0)
p1.mover_a(1, 2)
p2.mover_a(3, 4)
print(p2.distancia_a(p1))
print(p1.distancia_a(p2))
```

```
2.8284271247461903
2.8284271247461903
```

La mayoría de los lenguajes de programación orientados a objetos tienen el concepto de un **constructor**. Un método especial que crea e inicializa el objeto cuando se crea. Python es un poco diferente, tiene un constructor y un inicializador. La función de constructor rara vez se usa a menos que esté haciendo algo exótico. La función constructor se llama **__new__** y acepta exactamente un argumento; la clase que se está construyendo (se llama antes de que se construya el objeto, por lo que no hay argumento propio). También tiene que devolver el objeto recién creado. El método **__new__** tiene las siguientes características:

1. El método **__new__** se llama cuando la clase está lista para crear una

instancia.
2. El método **__new__** es siempre un método estático de la clase, incluso si no se agrega ningún decorador de método estático.

En Python, el método **__init__** es responsable de instanciar la instancia de la clase, y antes de que se llame a **__init__** , el método **__new__** decide si usar el método **__init__** o no, porque el método **__new__** puede llamar a otros constructores de clase o simplemente devolver otros objetos como instancias de esta clase.

El método **__del__** es un método especial, el cual se ejecuta al momento en el cual un objeto es descartado por el intérprete. El comportamiento de **__del__** es muy similar al **destructor** en otros lenguajes.

El método de inicialización de Python es el mismo que cualquier otro método, excepto que tiene un nombre especial, **__init__** .

> Los guiones bajos dobles iniciales y finales identifican un método especial que el intérprete de Python tratará como un caso especial.

In [6]:
```
class Prueba(object):

    def __new__(cls):
        print("NEW")
        return object.__new__(cls)

    def __init__(self):
        print("INIT")

test1 = Prueba()
```

```
NEW
INIT
```

Métodos de Clase

En ocasiones es necesario contar con métodos que interactúen con elementos de la clase de la cual el objeto es instanciado. Python permite definir métodos de clase para esto.

Los métodos de clase son aquellos que están ligados directamente con los atributos definidos en la clase que los contiene. Para definir un método de clase se utiliza el decorador **@classmethod** y por convención se utiliza **cls** como argumento inicial en lugar de **self** . Del mismo modo, los métodos de clase utilizan el prefijo **cls** para referirse a los atributos de la clase. Suelen utilizarse como métodos factoría.

In [7]:

```
import math

class Punto:
    origen_x = 0
    origen_y = 0

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

    @classmethod
    def punto_en_origen(cls):
        return cls(cls.origen_x, cls.origen_y)

    def mover_a(self, x, y):
        self.x = x
        self.y = y

    def distancia_a(self, otro_punto):
        return math.sqrt((self.x - otro_punto.x)**2 + (self.y - otro

p1 = Punto.punto_en_origen()
print(p1.origen_x, p1.origen_y, p1.x, p1.y)
```

```
0 0 0 0
```

Métodos estáticos

Los métodos estáticos hacen referencia a las instancias y métodos de una clase. Para definir un método estático se utiliza el decorador **`@staticmethod`** y no utiliza ningún argumento inicial.

Al no utilizar `self`, los métodos estáticos no pueden interactuar con los atributos y métodos de la instancia. Para referirse a los elementos de la clase, se debe utilizar el nombre de la clase como prefijo.

In [8]:

```python
import math

class Punto:
    origen_x = 0
    origen_y = 0

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

    @staticmethod
    def cambiar_origen(x, y):
        Punto.origen_x = x
        Punto.origen_y = y

    @classmethod
    def punto_en_origen(cls):
        return cls(cls.origen_x, cls.origen_y)

    def mover_a(self, x, y):
        self.x = x
        self.y = y

    def distancia_a(self, otro_punto):
        return math.sqrt((self.x - otro_punto.x)**2 + (self.y - otro

p1 = Punto(1, 1)
print(p1.origen_x, p1.origen_y, p1.x, p1.y)
Punto.cambiar_origen(2, 2)
print(p1.origen_x, p1.origen_y, p1.x, p1.y)
```

```
0 0 1 1
2 2 1 1
```

La diferencia entre un método estático y un método de clase es:

- El método estático no sabe nada sobre la clase y solo trata con los parámetros.
- El método de clase funciona con la clase ya que su parámetro siempre es la clase misma.

Sobrecarga de métodos

Es posible definir un método de manera que haya varias opciones para llamarlo, ha esta característica se le denomina **sobrecarga de métodos** y en Python se puede definir haciendo uso de métodos con parámetros predeterminados.

In [9]:

```python
import math

class Punto:
    origen_x = 0
    origen_y = 0

    def __init__(self, x=origen_x, y=origen_y):
        self.x = x
        self.y = y

    @staticmethod
    def cambiar_origen(x, y):
        Punto.origen_x = x
        Punto.origen_y = y

    @classmethod
    def punto_en_origen(cls):
        return cls(cls.origen_x, cls.origen_y)

    def mover_a(self, x=origen_x, y=origen_y):
        self.x = x
        self.y = y

    def distancia_a(self, otro_punto=None):
        if otro_punto is None:
            otro_punto = Punto.punto_en_origen()
        return math.sqrt((self.x - otro_punto.x)**2 + (self.y - otro

# __init__ sobrecargado
p1 = Punto(3, 5)
p2 = Punto(3)
p3 = Punto()
print(p1.x, p1.y)
print(p2.x, p2.y)
print(p3.x, p3.y)

# distancia_a sobrecargado
print(p1.distancia_a(p2))
print(p1.distancia_a())
```

```
3 5
3 0
0 0
5.0
5.830951894845301
```

Control de acceso

La mayoría de los lenguajes de programación orientados a objetos tienen un concepto de control de acceso. Esto está relacionado con la **abstracción**. Algunos atributos y métodos en un objeto están marcados como privados, lo que significa que solo ese objeto puede acceder a ellos. Otros están marcados como protegidos, lo que significa que solo esa clase y cualquier subclase tienen acceso. El resto son públicos, lo que significa que cualquier otro objeto puede acceder a ellos.

En Python todos los métodos y atributos de una clase están disponibles públicamente. Si queremos sugerir que un método no debe usarse públicamente, podemos poner una nota en la documentación que indique que el método está destinado sólo para uso interno (preferiblemente, con una explicación de cómo funciona la API pública).

Por convención, se utiliza un prefijo con un guión bajo `_` en el nombre del atributo o método. Los programadores de Python interpretarán esto como "*esta es una variable interna, piense tres veces antes de acceder a ella directamente*". Pero no hay nada dentro del intérprete que les impida acceder a él si creen que les conviene hacerlo.

Hay otra cosa que puede hacer para sugerir fuertemente que los objetos externos no accedan una propiedad o método: utilizar un prefijo con un doble guión bajo, `__` . Esto realizará el cambio de nombre en el atributo en cuestión (se ofusca el nombre). Básicamente, esto significa que los objetos externos aún pueden invocar el método si realmente quieren hacerlo, pero requiere un trabajo adicional y es un fuerte indicador de que se exige que el atributo permanezca privado. En estos casos es interesante utilizar métodos para recuperar el valor de los atributos ofuscados.

In [10]:
```
import math

class Punto:
    __origen_x = 0
    __origen_y = 0

    def __init__(self, x=__origen_x, y=__origen_y):
        self._x = x
        self._y = y

    @staticmethod
    def cambiar_origen(x, y):
        Punto.__origen_x = x
        Punto.__origen_y = y

    @staticmethod
    def obtener_origen_x():
        return Punto.__origen_x

    @staticmethod
    def obtener_origen_y():
        return Punto.__origen_y

    @classmethod
    def punto_en_origen(cls):
        return cls(cls.__origen_x, cls.__origen_y)

    def mover_a(self, x=__origen_x, y=__origen_y):
        self._x = x
        self._y = y

    def distancia_a(self, otro_punto=None):
        if otro_punto is None:
            otro_punto = Punto.punto_en_origen()
        return math.sqrt((self._x - otro_punto._x)**2 + (self._y - o

p1 = Punto(3, 9)
print(p1._x, p1._y)
```

```
3 9
```

In [11]:
```
print(p1.__origen_x, p1.__origen_y)
```

```
---------------------------------------------------------------------
-------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-11-d345d508b268> in <module>
----> 1 print(p1.__origen_x, p1.__origen_y)

AttributeError: 'Punto' object has no attribute '__origen_x'
```

In [12]:
```
print(p1._Punto__origen_x, p1._Punto__origen_y)
```

```
0 0
```

Para trabajar con variables "privadas" es habitual definir métodos para recuperar su información, modificarla o incluso para borrar la variable. En Python se puede

utilizar el decorador **`@property`** con métodos *getter*, *setter* y *deleter* asociados al nombre de la propiedad. De esta forma si se accede a la propiedad se ejecutará el método *getter*, si se modifica el *setter* y si se borra el *deleter*

In [13]:

```
class OtroPunto:
    # variables de clase compartida por todas las instancias
    __origen_x = 0
    __origen_y = 0

    def __init__(self, x, y):
        # variables de instancia única para cada instancia
        self.x = x
        self.y = y

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        if value >= 0:
            self._x = value
        else:
            self._x = 0

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, value):
        if value >= 0:
            self._y = value
        else:
            self._y = 0

# se ejecuta el método setter
p1 = OtroPunto(-1, 2)
print(p1.x, p1.y)
# se ejecuta el método setter
p1.x = -5
print(p1._x, p1._y)
# NO se ejecuta el método setter
p1._x = -5
print(p1._x, p1._y)
```

```
0 2
0 2
-5 2
```

Documentando el código

Python es un lenguaje de programación extremadamente fácil de leer; algunos podrían decir que es autodocumentado. Sin embargo, cuando se realiza una programación orientada a objetos, es importante escribir documentación API que resuma claramente lo que hace cada objeto y método. Mantener la documentación actualizada es difícil; la mejor manera de hacerlo es escribirlo directamente en nuestro código.

Python admite esto mediante el uso de docstrings. Cada encabezado de clase, función o método puede tener una cadena estándar de Python como la primera línea que sigue a la definición (la línea que termina en dos puntos). Esta línea debe tener sangría igual que el siguiente código.

Las cadenas de documentos son simplemente cadenas de Python encerradas con caracteres de apóstrofe (') o comillas ("). A menudo, las cadenas de documentos son bastante largas y abarcan varias líneas (la guía de estilo sugiere que la longitud de la línea no debe exceder los 80 caracteres), lo que puede estar formateado como cadenas de varias líneas, encerradas en un apóstrofe triple coincidente (''') o caracteres de comillas triples (""").

In [14]:

```
import math

class Punto:
    'Representa un punto en coordenadas geométricas bidimensionales'
    __origen_x = 0
    __origen_y = 0

    def __init__(self, x=__origen_x, y=__origen_y):
        '''Inicializa la posición de un nuevo punto.
           Se pueden especificar las coordenadas x e y.
           Si no se proporcionan el punto predeterminado es el orige
        self._x = x
        self._y = y

    def __repr__(self):
        'Define una cadena que representa un punto'
        return self.__class__.__name__ + "(x=%d, y=%d)" % (self._x,

    @staticmethod
    def cambiar_origen(x, y):
        'Permite cambiar el origen de coordenadas definido en la cla
        Punto.__origen_x = x
        Punto.__origen_y = y

    @staticmethod
    def obtener_origen_x():
        'Recupera el componente x del origen de coordenadas definido
        return Punto.__origen_x

    @staticmethod
    def obtener_origen_y():
        'Recupera el componente y del origen de coordenadas definido
        return Punto.__origen_y

    @classmethod
    def punto_en_origen(cls):
        "Crea un nuevo punto el origen de coordenadas definido"
        return cls(cls.__origen_x, cls.__origen_y)

    def mover_a(self, x=__origen_x, y=__origen_y):
        """Mueve el punto a una nueva ubicación en el espacio 2D.
        Si no se proporciona el punto de destino será el origen de c
        self._x = x
        self._y = y

    def distancia_a(self, otro_punto=None):
        """Calcula la distancia desde este punto a un segundo punto
Si no se proporciona el segundo punto se toma el origen de coordenad
Esta función utiliza el teorema de Pitágoras para calcular la distan
La distancia es devuelta como un float."""
        if otro_punto is None:
            otro_punto = Punto.punto_en_origen()
        return math.sqrt((self._x - otro_punto._x)**2 + (self._y - o

print(Punto.__doc__)
print("-----------------------")
print(Punto.distancia_a.__doc__)
```

```
Representa un punto en coordenadas geométricas bidimensionales
-----------------------
```

```
Calcula la distancia desde este punto a un segundo punto pasado como
parámetro.
Si no se proporciona el segundo punto se toma el origen de coordenad
as
Esta función utiliza el teorema de Pitágoras para calcular la distan
cia entre los dos puntos.
La distancia es devuelta como un float.
```

La función Python `__repr__` devuelve la representación del objeto. Podría ser cualquier expresión válida de Python como una tupla, diccionario, cadena, etc. Este método se llama cuando se invoca la función `repr` en el objeto, en ese caso, la función `__repr__` debe devolver una cadena, de lo contrario se generará un error.

In [15]:
```
p1 = Punto(3, 3)
repr(p1)
```

Out[15]:
```
'Punto(x=3, y=3)'
```

Objetos (instancias)

Son una pieza de datos encapsulados con funcionalidades en un programa Python que se construye de acuerdo con una definición de clase. A menudo, un objeto corresponde a una cosa en el mundo real. Un objeto consiste en un número arbitrario de atributos y métodos, encapsulados dentro de una sola unidad. Resumiendo, un objeto es una colección de datos con comportamientos asociados.

Los objetos son instancias de clases que se pueden asociar entre sí. Una instancia de objeto es un objeto específico con su propio conjunto de datos y comportamientos. El proceso de crear un objeto a partir de una clase se denomina instanciación.

El propósito clave de modelar un objeto en un diseño orientado a objetos es determinar cuál será la interfaz pública de ese objeto. La interfaz es la colección de atributos y métodos que otros objetos pueden usar para interactuar con ese objeto. No necesitan, y a menudo no se les permite, acceder al funcionamiento interno del objeto.

Un ejemplo común del mundo real es la televisión. Nuestra interfaz con la televisión es el control remoto. Cada botón del control remoto representa un método que se puede invocar en el objeto de televisión. Cuando nosotros, como objeto de llamada, accedemos a estos métodos, no sabemos ni nos importa si el televisor recibe su señal de una antena, una conexión de cable o una antena parabólica. No nos importa qué señales electrónicas se envían para ajustar el volumen, o si la imagen está preprocesada.

Este proceso de ocultar la implementación, o detalles funcionales, de un objeto

se denomina **ocultación de información**. A veces también se conoce como **encapsulación**, pero la encapsulación es en realidad un término que lo abarca todo. La distinción entre encapsulación y ocultación de información es en gran medida irrelevante, especialmente a nivel de diseño. Muchas referencias prácticas usan estos términos indistintamente. Como programadores de Python, en realidad no tenemos o necesitamos ocultar información verdadera, por lo que la definición más amplia para la encapsulación es adecuada.

La interfaz pública, sin embargo, es muy importante. Necesita ser cuidadosamente diseñada ya que es difícil cambiarla en el futuro. Cambiar la interfaz interrumpirá los objetos del cliente que la estén llamando. Podemos cambiar las partes internas todo lo que queramos, por ejemplo, para hacerlo más eficiente, o para acceder a los datos a través de la red y localmente, y los objetos del cliente aún podrán hablar con él, sin modificaciones, utilizando el público interfaz. Pero si cambiamos la interfaz cambiando los nombres de los atributos a los que se accede públicamente, o alterando el orden o los tipos de argumentos que un método puede aceptar, todos los objetos del cliente también tendrán que modificarse.

La **abstracción** es otro concepto orientado a objetos relacionado con la encapsulación y la ocultación de información. En pocas palabras, la abstracción significa tratar con el nivel de detalle más apropiado para una tarea determinada. Es el proceso de extraer una interfaz pública de los detalles internos. El conductor de un automóvil necesita interactuar con la dirección, el acelerador y los frenos. El funcionamiento del motor, el tren de transmisión y el subsistema de frenos no le importan al conductor. Un mecánico, por otro lado, trabaja en un nivel diferente de abstracción, ajusta el motor y desangra los frenos. Este es un ejemplo de dos niveles de abstracción para un automóvil.

Principios básicos de POO

En particular, cubriremos la sintaxis y los principios de Python para:

- Herencia básica
- Herencia de componentes internos
- Herencia múltiple
- Polimorfismo y tipificación dinámica (duck typing).

Herencia básica

El uso más simple y obvio de la herencia es agregar funcionalidad a una clase existente. Técnicamente, cada clase que creamos usa herencia. Todas las clases de Python son subclases de la clase especial denominada `object`. Esta clase proporciona muy poco en términos de datos y comportamientos (los comportamientos que proporciona son todos métodos de doble subrayado

destinados solo para uso interno), pero permite que Python trate todos los objetos de la misma manera.

Si no heredamos explícitamente de una clase diferente, nuestras clases heredarán automáticamente de `object`. Sin embargo, podemos afirmar abiertamente que nuestra clase deriva del objeto utilizando la siguiente sintaxis:

```
class MiClase(object):
    pass
```

Python 3 hereda automáticamente de `object` si no proporcionamos explícitamente una superclase diferente. Una superclase, o clase padre, es una clase de la que se hereda. Una subclase es una clase que hereda de una superclase.

In [16]:
```
class Punto3D(Punto):
    pass
```

Python tiene dos funciones integradas que funcionan con herencia:

- `isinstance` para verificar el tipo de una instancia: `isinstance (obj, int)` será True solo si `obj .__ class__` es `int` o alguna clase derivada de `int`.
- Use `issubclass` para verificar la herencia de la clase: `issubclass (bool, int)` es `True` ya que `bool` es una subclase de `int`.

In [17]:
```
p1 = Punto()
print(isinstance(p1, Punto))
print(issubclass(Punto3D, Punto))
```

```
True
True
```

Sobreescritura y super

La herencia es excelente para agregar un nuevo comportamiento a las clases existentes, pero ¿qué pasa con el cambio de comportamiento?. Si queremos crear una tercera variable para definir el eje `z` y que esté disponible en la inicialización, tenemos que reescribir `__init__`. Reescribir significa alterar o reemplazar un método de la superclase con un nuevo método (con el mismo nombre) en la subclase. No se necesita una sintaxis especial para hacer esto. El método recién creado de la subclase se llama automáticamente en lugar del método de la superclase. Por ejemplo:

In [18]:
```
class Punto3D(Punto):
    __origen_x = 0
    __origen_y = 0
    __origen_z = 0

    def __init__(self, x = __origen_x, y = __origen_y, z = __origen_
        self._x = x
        self._y = y
        self._z = z
```

Se puede reescribir cualquier método, no solo `__init__` . Sin embargo, antes de continuar, debemos abordar algunos problemas en este ejemplo. Nuestras clases `Punto` y `Punto3D` tienen un código duplicado para configurar las propiedades `x` e `y` . Esto puede complicar el mantenimiento del código, ya que tenemos que actualizar el código en dos o más lugares.

Lo que realmente necesitamos es una forma de ejecutar el método original `__init__` en la clase `Punto` . Esto es lo que hace la función **`super`** que devuelve el objeto que es una instancia de la clase padre, lo que nos permite llamar al método del padre directamente:

In [19]:
```
class Punto3D(Punto):
    __origen_z = 0

    def __init__(self, x = Punto.obtener_origen_x(), y = Punto.obten
        super().__init__(x, y)
        self._z = z
```

> En Python 2, la sintaxis sería: `super(Punto3D, self).__init__()` . Observar que el primer argumento es el nombre de la clase hija, no el nombre de la clase padre a la que desea llamar.

Se puede hacer una llamada `super` dentro de cualquier método, no solo `__init__` . Esto significa que todos los métodos pueden modificarse mediante reescritura y llamadas a `super` . La llamada a `super` también se puede hacer en cualquier punto del método; no tenemos que hacer la llamada como la primera línea del método. Por ejemplo, es posible que necesitemos manipular o validar los parámetros entrantes antes de reenviarlos a la superclase.

In [20]:

```
import math

class Punto3D(Punto):
    'Representa un punto en coordenadas geométricas bidimensionales'
    __origen_z = 0

    def __init__(self, x = Punto.obtener_origen_x(), y = Punto.obten
        '''Inicializar la posición de un nuevo punto.
           Se pueden especificar las coordenadas x, y, z.
           Si no se proporcionan el punto predeterminado es el orige
        super().__init__(x, y)
        self._z = z

    def __repr__(self):
        'Define una cadena que representa un punto'
        return super().__repr__()[0:-1] + ", z=%d)" % (self._z)

    @staticmethod
    def cambiar_origen(x, y, z):
        'Permite cambiar el origen de coordenadas definido en la cla
        Punto.cambiar_origen(x, y)
        __origen_z = z

    @staticmethod
    def obtener_origen_z():
        'Recupera el origen de coordenadas definido en la clase'
        return Punto3D.__origen_z

    @classmethod
    def punto_en_origen(cls):
        "Crea un nuevo punto el origen de coordenadas definido"
        return cls(Punto.obtener_origen_x(), Punto.obtener_origen_y(

    def mover_a(self, x = Punto.obtener_origen_x(), y = Punto.obtene
        """Mueve el punto a una nueva ubicación en el espacio 2D.
        Si no se proporciona el punto de destino será el origen de c
        super().mover_a(x, y)
        self._z = z

    def distancia_a(self, otro_punto=None):
        """Calcula la distancia desde este punto a un segundo punto
        Si no se proporciona el segundo punto se toma el origen de c
        Esta función utiliza el teorema de Pitágoras para calcular l
        La distancia es devuelta como un float."""
        if otro_punto is None:
            otro_punto = Punto3D.punto_en_origen()
        return math.sqrt((self._x - otro_punto._x)**2 + (self._y - o

p1 = Punto3D(1,2,3)
p2 = Punto3D(4)
p3 = Punto3D.punto_en_origen()
print(repr(p1))
print(repr(p2))
print(repr(p3))
print(p1.distancia_a(p2))
```

```
Punto3D(x=1, y=2, z=3)
Punto3D(x=4, y=0, z=0)
Punto3D(x=0, y=0, z=0)
4.69041575982343
```

Extendiendo clases de Python

Un uso interesante de este tipo de herencia es agregar funcionalidad a las clases integradas en Python. Los elementos integrados comúnmente extendidos son `object`, `list`, `set`, `dict`, `file` y `str`. También se pueden heredar los tipos numéricos como `int` y `float`.

In [21]:
```
class ListaPuntos(list):
    def buscar_punto(self, otro_punto):
        for punto in self:
            if punto.__class__.__name__ == otro_punto.__class__.__na
                if hasattr(punto, 'z'):
                    if punto._x == otro_punto._x and punto._y == otr
                        return True
                else:
                    if punto._x == otro_punto._x and punto._y == otr
                        return True
        return False

lista = ListaPuntos()
lista.append(Punto())
lista.append(Punto(1, 2))
lista.append(Punto3D(3, 4, 5))
print(lista)
print(lista.buscar_punto(Punto(1, 2)))
print(lista.buscar_punto(Punto3D(1, 2, 5)))
print(lista.pop())
print(lista)
```

```
[Punto(x=0, y=0), Punto(x=1, y=2), Punto3D(x=3, y=4, z=5)]
True
False
Punto3D(x=3, y=4, z=5)
[Punto(x=0, y=0), Punto(x=1, y=2)]
```

Herencia múltiple

La herencia múltiple es un tema delicado. En principio, es muy simple: una subclase que hereda de más de una clase padre puede acceder a la funcionalidad de ambos. En la práctica, esto es menos útil de lo que parece y muchos programadores expertos recomiendan no usarlo. La herencia múltiple funciona bien cuando se mezclan métodos de diferentes clases, pero se vuelve muy complicado cuando tenemos que llamar a métodos en la superclase. Hay múltiples superclases.

La sintaxis para la herencia múltiple se parece a una lista de parámetros en la definición de clase. En lugar de incluir una clase base dentro de los paréntesis, incluimos dos (o más), separados por una coma. En Python la jerarquía de clases se define de derecha a izquierda, de este modo la clase de la derecha es la clase base que se extiende con las clases de la izquierda. Pero si se sobreescriben métodos o propiedades en las clases, la prioridad de cómo se resuelven los

métodos es de izquierda a derecha.

La forma más simple y útil de herencia múltiple se llama **mixin**. Un mixin es generalmente una superclase que no debe existir por sí sola, sino que debe ser heredada por alguna otra clase para proporcionar una funcionalidad adicional.

In [22]:
```
class CBase(object):
    def test(self):
        print("Clase Base")

class C1(object):
    def test(self):
        print("Clase 1")

class C2(object):
    def test(self):
        print("Clase 2")

class MiClase(C2, C1, CBase):
    pass

obj = MiClase()
obj.test()
```

```
Clase 2
```

El orden en que se pueden invocar los métodos se puede adaptar en la y modificando el atributo `__mro__` (Method Resolution Order) en la clase.

In [23]:
```
MiClase.__mro__
```

Out[23]:
```
(__main__.MiClase, __main__.C2, __main__.C1, __main__.CBase, object)
```

Polimorfismo y Tipificación dinámica

El polimorfismo describe un concepto simple: diferentes comportamientos suceden dependiendo de qué subclase se esté utilizando, sin tener que saber explícitamente de qué tipo es realmente la subclase.

In [24]:
```
lista = ListaPuntos()
lista.append(Punto(3, 4))
lista.append(Punto(6, 8))
lista.append(Punto3D(9, 12, 3))
print(lista)
for punto in lista:
    print(punto.distancia_a())
```

```
[Punto(x=3, y=4), Punto(x=6, y=8), Punto3D(x=9, y=12, z=3)]
5.0
10.0
15.297058540778355
```

En los lenguajes de programación orientados a objetos, se conoce como "*duck typing*" el estilo de tipificación dinámica de datos en que el conjunto actual de métodos y propiedades determina la validez semántica, en vez de que lo hagan

la herencia de una clase en particular o la implementación de una interfaz específica.

El polimorfismo es una de las razones más importantes para usar la herencia en muchos contextos orientados a objetos. Debido a que cualquier objeto que proporcione la interfaz correcta se puede usar indistintamente en Python, reduce la necesidad de superclases comunes polimórficas. La herencia aún puede ser útil para compartir código, pero, si todo lo que se comparte es la interfaz pública, lo único que se requiere la tipificación dinámica. Esta necesidad reducida de herencia también reduce la necesidad de herencia múltiple; a menudo, cuando la herencia múltiple parece ser una solución válida, podemos usar la tipificación dinámica para imitar una de las múltiples superclases.

Otra característica útil de la tipificación dinámica es que el objeto tipado solo necesita proporcionar los métodos y atributos a los que realmente se está accediendo.

In [25]:
```
class MiPunto2D:
    def __init__(self, x=0, y=0):
        self._x = x
        self._y = y

    def distancia_a(self, otro_punto=None):
        if otro_punto is None:
            otro_punto = Punto.punto_en_origen()
        return math.sqrt((self._x - otro_punto._x)**2 + (self._y - o

lista.append(MiPunto2D(9, 12))
for punto in lista:
    print(punto.distancia_a())
```

```
5.0
10.0
15.297058540778355
15.0
```