Argenitna Programa 4.0

Universidad Nacional de San Luis

Desarrollador Python

Lenguaje de Programación Python: Soporte Orientado a Objetos

Autor:

Dr. Mario Marcelo Berón

UNIDAD 6

LENGUAJE DE PROGRAMACIÓN PYTHON: SOPORTE ORIENTADO A OBJETOS

6.1. Introducción

En las unidades anteriores se usaron objetos pero el estilo de programación fue procedural. Python es un lenguaje multiparadigma dado que permite programar usando el paradigma imperativo, orientado a objeto y funcional o una mezcla de estilos dado a que no fuerza a programar en ningún estilo específico. Es perfectamente posible escribir programas en el estilo procedural y para programas pequeños raramente esta forma de abordar el problema causará inconvenientes. No obstante, a medida que el tamaño del programa crece, el *Paradigma Orientado a Objetos* proporciona ventajas que el programador puede aprovechar. En esta unidad se abordan los conceptos y técnicas fundamentales para programar orientado a objetos en Python. La primer sección está dedicada para los programadores principiantes y para aquellos que han hecho programación procedural. La sección comienza dado a conocer algunos problemas que pueden surgir con la programación procedural que el

paradigma orientado a objetos puede resolver. Luego presenta una solución utilizando Python orientado a objetos y explica la terminología relevante.

La segunda sección aborda la creación de tipos de datos personalizados que almacenan datos simples y la tercera sección trata la creación de tipos de dato colección que pueden almacenar cualquier número de objetos de cualquier tipo.

6.2. La Aproximación Orientada a Objetos

En esta sección se presentan algunos problemas que aparecen en un enfoque puramente prodecural considerando una situación donde se quieren representar círculos, potencialmente muchos de ellos. El dato mínimo requerido para representar un círculo es su posición y su radio.

Una posibilidad es usar una triupla para cada uno de ellos, por ejemplo:

```
círculo = (11, 60, 8)
```

Una desventaja de esta forma es que no queda claro que representa cada elemento de la triupla. Podría significar (x,y,radio) o (radio,x,y). Otra desventaja es que se puede acceder a los elementos por índices de posición solamente. Si se tienen dos funciones *distanciaDesdeElOrigen(x,y)* y *distanciaAlBordeDesdeElOrigen(x,y,radio)* se necesitaría usar desempaquetamiento de tuplas para llamarlas con una tupla que representa a un círculo:

```
distancia = distanciaDesdeElOrigen(*círculo[:2])
distancia = distanciaAlBordeDesdeElOrigen(*círculo)
```

Ambas funciones asumen que las tuplas son de la forma (x,y,radio).

Se puede resolver el problema anterior usando una tupla nombrada.

```
import collections
Círculo = collections.namedtuple("Círculo", "x y radio")
circulo = Círculo(13, 84, 9)
distancia = distanciaDesdeElOrigen(círculo.x, círculo.y)
```

Esto permite crear círculos (3-uplas) con atributos con nombres lo cual hace las llamadas a las funciones mucho más fáciles de entender dado que para acceder as sus elementos se pueden usar sus nombres. Desafortunadamente queda un problema, no hay nada que impida crear un círculo inválido:

```
círculo = Círculo(33, 56, -5)
```

No tiene sentido tener un círculo con radio negativo. No obstante, la tupla nombrada se puede crear sin disparar una excepción. El error se notaría cuando se invoque a la función *distanciaAlBordeDesdeElOrigen()* solo si la función verifica que el radio no sea negativo. Esta imposibilidad de validar cuando se crea un objeto es quizás el peor aspecto de tomar una aproximación puramente procedural. Cuando se crea un círculo no hay nada que imposibilite colocar un dato inválido.

Si se desea que el círculo sea mutable y que se pueda mover cambiando sus coordenadas y cambiar el tamaño modificando su radio, se puede hacer usando el método privado *collections.namedtuple._replace()*.

```
círculo= circulo._replace(radio=12)
```

Si los círculos necesitan muchos cambios quizás lo mejor sea usar un tipo de dato mutable tal como una lista:

```
círculo = [36, 77, 8]
```

Esto no da protección respecto de crear un círculo inválido. Además, lo mejor que se puede hacer para acceder a sus ítem es crear constantes: círculo[RADIO]=5. Pero usar una lista genera problemas adicionales, por ejemplo se puede llamar a *círculo.sort()*.

El uso de un diccionario puede ser otra alternativa por ejemplo: círculo = dict(x=36, y=77, radio=8), pero nuevamente no hay una forma de asegurar un radio válido y no hay una forma de prevenir que métodos inapropiados se llamen.

Comentario

En realidad cuando se está escribiendo un programa utilizando el Paradigma de Programación Imperativo, el programador debe intentar usar el concepto de Tipo de Dato Abstracto. Un Tipo de Dato Abstracto se define como un conjunto de datos y operaciones asociadas donde la única forma de acceder a los datos es a través de esas operaciones. Programar usando la idea antes mencionada no solo organiza de una mejor manera el código sino que además simplifica el proceso de transformar un programa escrito con el Paradigma Imperativo al mismo programa pero orientado a objetos. Esto se debe a que un Tipo de Dato Abstracto se puede implementar con una clase.

Cuando se programa usando esta aproximación, no se subsanan las deficiencias mencionadas con anterioridad, ni tampoco se lleva a cabo la efectiva *encapsulación* de los datos. Es decir no se impide que los datos sean accedidos de otra forma sin usar las operaciones. No obstante, se gana en modularidad, claridad conceptual, organización del código, localización de errores, entre otras tantas ventajas.

Tanto las deficiencias mencionadas como la encapsulación se deben llevar a cabo por *convención*. Es decir el programador acuerda seguir ciertas reglas con el propósito de mantener las ventajas antes mencionadas.

6.3. Conceptos y Terminología

Lo que se requiere es una forma de empaquetar los datos que representan a un círculo, y alguna forma de restringir los métodos que se pueden aplicar a los datos de forma tal que solo operaciones válidas se puedan aplicar a los datos. Ambos requerimientos se pueden alcanzar definiendo un tipo de dato *Círculo*.

En esta unidad se verá como crear un tipo de datos pero antes de hacerlo es necesario introducir algunos conceptos preliminares y explicar la terminología que se usará.

Se empleará *Clase* y *Tipo de Dato* como sinónimos[1] [2]. En Python es posible crear clases personalizadas que se integran completamente y que pueden ser usadas como tipos primitivos. Se han descrito varias clases como *dict*, *int*, *str*, etc. Se usará el término *objeto* y ocasionalmente *instancia* para hacer referencia a una instancia de una clase en particular. Por ejemplo, 5 es un objeto de *int*.

Las clases encapsulan los datos y las operaciones que se pueden aplicar sobre los datos[3]. Por ejemplo, la clase *str* almacena una cadena de caracteres escritas en unicode y soporta la operación *str.upper()*. Muchas clases también tienen características adicionales, por ejemplo, se pueden concatenar cadenas de caracteres (o secuencias) con el operador +, o encontrar la longitud de una secuencia utilizando *len()*. Tales características son provistas por métodos especiales los cuales son similares a los métodos excepto que ellos comienzan con dos guiones bajos y están predefinidos. Por ejemplo si se desea crear una clase que contenga el operador + y la función *len()* se tienen que implementar los métodos especiales $\_\_add\_\_()$ y $\_\_len\_\_()$ en la clase. Inversamente, nunca se definiría un método que comience y finalice con dos guiones bajos al menos que sea un método especial. Esto asegurará que no se entrará en conflicto con versiones anteriores de Python. Los objetos usualmente tienen atributos los métodos son atributos llamables y los otros son datos. Por ejemplo, *complex* tiene atributos *real* e *imag* y muchos métodos, incluyendo métodos especiales como $\_\_add\_\_()$, $\_\_sub\_\_()$ (para que soporten los operadores binarios + y -) y métodos normales como *conjugate()*.

Los atributos de dato (frecuentemente referenciados como atributos[4]) son normalmente implementados como *variables de instancia*, es decir variables que pertenecen a un objeto en particular. Se verán ejemplos de esto y también como proveer atributos de datos como propiedades. Una *propiedad* es un ítem

[1] Esta afirmación no es estrictamente cierta dado que los lenguajes orientados a objetos permiten ciertas flexibilizaciones en lo que a encapsulación se refiere. Por ejemplo en lenguajes como Java es posible definir variables de instancia (atributos) cuyo modificador de alcance es público o protegido.

[2] En el caso de Python todas las variables son públicas y el cambio en el alcance está dado por convención.

[3] Una clase es un Tipo de Dato Abstracto.

[4] También se conocen en la jerga como *Variables de Instancia*.

de un objeto que puede ser accedido como una variable de instancia pero que es administrado con un método por detrás. El uso de propiedades hace más fácil la validación de datos.

Dentro de un método (el cual es como una función cuyo primer argumento es la instancia[5] sobre la cual operará) diferentes clases de variables son potencialmente accesibles. Las variables de instancia se pueden acceder cualificando[6] su nombre con el nombre de la instancia. Las *variables de clase* (algunas veces llamadas *variables estáticas*) pueden ser accedidas cualificando[7] el nombre con el nombre de la clase y las variables globales, es decir la variables del módulo, pueden ser accedidas sin cualificación.

Algunas bibliografías de Python usan el conceto de *espacio de nombres*, el cual se entiende como un mapeo de nombres a objetos. Los módulos son espacios de nombres, por ejemplo después de la sentencia *import math* se pueden acceder a objetos definidos en el módulo *math* cualificando con el nombre del espacio de nombres, por ejemplo *math.pi* y *math.sin()*. De la misma manera una clase y un objeto son también espacios de nombres. Por ejemplo, si se tiene z=complex(1,2), el espacio de nombres objeto z tiene dos atributos a los cuales se pueden acceder los cuales son *real* e *imag* (z.real y z.imag).

Una de las ventajas de la orientación a objetos es la posibilidad de especializar una clase. Es decir se puede hacer una nueva clase que herede todos los atributos (datos y métodos) de la clase original, y es posible agregar nuevos métodos y variables y reemplazar otros.

Se puede crear una *subclase* (otro término para *especializar*) cualquier clase de Python, sea primitiva, de la librería estándar o creada por el programador.

La posibilidad de crear subclases es una de las principales ventajas del *Paradigma de Programación Orientada a Objetos* dado que permite, de manera directa, usar una clase existente que ya ha sido probada como la base para crear una clase que extiende la original. Incorporar atributos o nue-

[5]También se conoce con el nombre de *Objeto Receptor*.
[6]Anteponiendo el nombre de la instancia y luego el nombre de la variable.
[7]Anteponiendo el nombre de la clase al nombre de la variable.

vas funcionalidades es claro y directo. Además,es posible pasar objetos de la nueva clase como parámetros de funciones y métodos de la clase original.

Se utiliza el término *Clase Base* para hacer referencia a una clase que es heredada. Una clase base puede ser un ancestro directo o puede encontrarse más arriba en el *Árbol de Herencia*. Otro término usado para la clase base es *super clase*.

Se utiliza el término *subclase*, *Clase Derivada* o *Derivada* para hacer referencia a la clase que hereda de otra clase.

En Python toda clase primitiva o de una librería o toda clase que crea el programador deriva directamente o indirectamente de la última clase base denominada *object*.

Cualquier método puede ser *sobre-escrito* esto es re-implementado en una subclase. Si se tiene un objeto de una clase *MiDict* la cual hereda de *dict* y se invoca a un método que está definido en ambas clases, Python correctamente invocará al métdo de *MiDict* esto se conoce con el combre de *ligadura de métodos dinámica* también conocido como *polimorfismo*. Si se necesita llamar a la versión de la super clase dentro de la re-implementación del método se puede hacer usando la función primitiva *super()*.

Python tiene soporte para el *tipado del pato* (*duck typing*) esto es: *si camina como un pato y emite sonido como un pato entonces es un pato*. En otras palabras, si se desea llamar a cierto método sobre un objeto, no importa de que clase es el objeto, solamente el objeto tiene que tener los métodos que se desean llamar.

La herencia se utiliza para modelar relaciones *es un* esto es objetos que son esencialmente lo mismo que algunos objetos de otras clases, pero con algunas variantes, tales como datos extras o métodos extras. Otro enfoque es usar *agregación* (también llamada *composición*), esto es donde una clase incluye una o más variables de instancias que son de otras clases. La *agregación* se utiliza para modelar las relaciones *tiene una*. En Python, toda clase usa herencia porque todas las clases tienen como última clase base a *object*, y muchas clases usan agregación ya que tienen variables de instancias de diferentes tipos.

Algunos lenguajes orientados a objetos tienen dos características que

Python no posee. La primera es la *sobrecarga* esto consiste en tener métodos con el mismo nombre pero con diferente número de parámetros en la misma clase. Gracias a la versatilidad del manejo de parámetros de Python nunca esto fue una limitación en la práctica. La segunda característica es el *Control de Acceso* es decir no tiene un mecanismo que permita lograr la privacidad de los datos[8][9]. Siempre que se crea una variable de instancia o método que comienza con dos guión bajos, Python informará el acceso no intencional dado que dichos atributos se consideran privados.

De la misma forma que se utilizan letras mayúsculas para la primer letra de los módulos las mismas se usan para las clases personalizadas. Se pueden definir tantas clases como se desee sea en el programa principal o en módulos. Los nombres de las clases no tienen que concordar con el nombre del módulo y los módulos pueden contener tantas clases como se desee.

6.3.1. Ejercicios

Ejercicio 1: Defina y explique los siguientes conceptos:

1. Tipo de Dato Abstracto
2. Encapsulación
3. Clase
4. Variable de Insntancia
5. Método de Instancia
6. Clase Base
7. Subclase
8. Herencia
9. Árbol de Herencia
10. Duck Typing

[8]Esta es una característica que disminuye significativamente la potencia del soporte orientado a objetos del lenguaje. Esto se debe a que por convención se mantiene la encapsulación.

[9]Al no poseer modificadores de acceso no se puede restringir el acceso a los datos y métodos heredados dado que todo es público.

Ejercicio 2: Implemente el tipo de dato abstracto Persona. Los datos que se desean registrar de una persona son: el nombre y la edad.

Ejercicio 3: Escriba un programa principal que:

1. Inserte en una lista n personas.
2. Almacne en una lista la/s persona/s de mayor edad.
3. Imprima la lista.

Nota: Luego de haber resuelto este ejercicio reflexione respecto de las ventajas de haberlo realizado usando el concepto de Tipo de Dato Abstracto.

Ejercicio 4: Muestre diferentes formas de violar la encapsulación del tipo de dato abstracto Persona.

6.4. Clases Personalizadas

A continuación se muestra la sintaxis para crear clases.

Clase Punto

```
class NombreDeLaClase:
 cuerpo


class NombreDeLaClase(ClaseBase):
 cuerpo
```

class es una sentencia con la cual se pueden crear clases. Los métodos de la clase se crean usando sentencias *def* en el cuerpo. Las instancias de la clase se crean invocando a la clase con los parámetros necesarios, por ejemplo x=complex(4,8) crea un número complejo e inicializa x con una referencia al objeto.

Clase Punto

```
class Punto:
 pass
```

Se crea una clase Punto la cual no tiene ni variables de instancia ni métodos de instancia.

Creción de Instancia de Punto

```
p=Punto()
```

Se crea una instancia de la clase punto. Observe que se coloca el nombre de la clase y los parámetros que se requieren. En este momento la creación de instancias no tiene parámetros dado que por ahora la clase Punto no realiza ninguna tarea.

6.4.1. Atributos y Métodos

Para iniciar esta sección se implementará una clase *Punto* la cual permite registrar las coordenadas x e y en el plano. La implementación se muestra se a continuación.

Clase Punto

```
class Punto:
 def __init__(self, x=0, y=0):
  self.x = x
  self.y = y

 def distanciaDesdeElOrigen(self):
  return math.hypot(self.x, self.y)
```


Clase Punto-Continuación

```
def __eq__(self, otro):
 return self.x == otro.x and self.y == otro.y

def __repr__(self):
 return "Punto({0.x!r}, {0.y!r})".format(self)

def __str__(self):
 return "({0.x!r}, {0.y!r})".format(self)
```

Dado que no se especifica ninguna subclase, *Punto* es una subclase[10] de *object*[11]. La clase Punto tiene dos atributos *self.x* y *self.y* y cinco métodos sin contar los heredados, cuatro de ellos son métodos especiales. Una vez que se importa el módulo *Forma*[12] la clase *Punto* se puede usar como cualquier otra. Los atributos se pueden acceder directamente por ejemplo: y=a.y[13] y la clase se integra bien con el resto de las clase de Python. Provee soporte para el operador de igualdad (==) y para producir cadenas de caracteres que representan el objeto, entre otras tantas operaciones. Python es lo suficientemente inteligente y provee el operador de desigualdad (!=) en base al operador de igualdad[14]. Python proporciona automáticamente el primer argumento en los llamados a métodos el cual es una referencia al objeto en sí mismo (usualmente conocido con el nombre de *self*). Este argumento se incluye en la lista de parámetros del método y por convención se lo denomina *self*.

[10] El concepto de subclase se retomará cuando se estudie *Herencia*

[11] Es la clase raíz de la jerarquía de clases proporcionada por Python.

[12] Forma es el nombre del archivo .py donde se encuentra definida la clase Punto.

[13] Observe que esta forma de acceder viola la encapsulación. Esto se debe los atributos en Python son públicos.

[14] Es posible especificar cada operador individualmente se se desea tener el control total, por ejemplo, si no no son opuestos exactos

Uso de la Clase Punto

```
import Forma
a = Forma.Punto()
repr(a)    # retorna: 'Punto(0, 0)'
b = Forma.Punto(3, 4)
str(b)     # retorna: '(3, 4)'
b.distanciaDesdeElOrigen() # retorna: 5.0
b.x = -19 #Se rompe la encapsulación.
          #No implementa un TDA
str(b)     # retorna: '(-19, 4)'
a == b, a != b  # retorna: (False, True)
```

Todos los atributos del objeto (datos y métodos) deben ser cualificados con *self*[15]. Esta característica implica un poco más de escritura comparado con otros lenguajes, pero tiene la ventaja de proveer claridad: *Siempre se conoce que se está accediendo a un atributo del objeto si éste está cualificado con self.*

Para crear un objeto se realizan los siguientes pasos:

1. Se debe crear un objeto crudo o no inicializado.

2. El objeto se debe inicializar para que se pueda usar.

Algunos lenguajes combinan esos dos pasos en uno pero Python los separa. Cuando un objeto se crea (p=Forma.Punto()), primero se invoca el método *__new()__* el cual crea el objeto, y luego se invoca el método especial *__init__()* para inicializarlo.

En la práctica casi toda clase que se implementa redefine solo el método *__init__()* ya que el método *__new__()* es casi siempre suficiente y se invoca automáticamente si no se provee una nueva implementación para él. No tener que reimplementar métodos en una subclase es otro beneficio de la Programación Orientada a Objetos: *Si los métodos de la super clase son*

[15]Recuerde que acceder a los datos directamente rompe la encapsulación.

suficientes no se tienen que reimplementar en la subclase. Cuando se invoca un método sobre un objeto si la clase del objeto no implementa dicho método, Python lo busca en la super clase y sus super clases y así siguiendo hasta encontrarlo o en su defecto se disparará una excepción.

Por ejemplo, si se ejecuta *p=Forma.Punto()*, Python comienza a buscar el método Punto.__new__()[16]. Dado que la clase no tiene implementado este método Python continúa su búsqueda en la super clase. En este caso, hay solo una super clase, *object*, y tiene el método requerido, en consecuencia se ejecuta *objet.__new__()* y crea un objeto sin inicializar. Luego Python busca el inicializador *__init__()* ya que la clase lo reimplementa, no se necesita ir a buscarlo y por lo tanto se invoca *Punto.__init__()*. Finalmente, *p* es una referencia a un objeto recientemente creado e inicializado de tipo *Punto*.

En lo que sigue se explicarán en detalle cada uno de los métodos que se han implementado en la clase *Punto*.

__init__()

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

Las dos variables de instancias *self.x* y *self.y* se crean en el inicializador y se le asginan los valores de los parámetros *x* e *y*. Debido a que Python encuentra el inicializador cuando se crea un objeto *Punto*, el método *object.__init__()* no se invoca dado que está redefinido en la subclase.

Los puristas de la orientación a objetos comienzan el método con un llamado al método *__init__()* de la clase base es decir ejecuta un *super.__init__()*. El efecto de la sentencia anterior es invocar a *__init__()* de la clase base. Para las clases que heredan directamente de *object* no es necesario llamar al método *__init__()*. La invocación a éste método se necesita cuando cuando se diseña una sublclase o cuando se crea una clase que

[16]La búsqueda se comienza en la clase del objeto receptor del mensaje.

no hereda directamente de *object*.

distanciaDesdeElOrigen()

```
def distanciaDesdeElOrigen(self):
 return math.hypot(self.x, self.y)
```

Este es un método convencional y realiza una computación basada en las variables de instancia. Es común para los métodos cortos tener solo el objeto receptor como único argumento ya que todos los datos que el método necesita están dentro del objeto.

Los métodos no deben tener nombres que comiencen y finalicen con dos guiones bajos a no ser que sean uno de los métodos especiales. Python provee métodos especiales para todos los operadores de comparación.

Todas las instancias de las clases personalizadas soportan el operador == por defecto y la comparación retorna como resultado False a no ser que se compare con el objeto en sí mismo. Se puede sobre escribir este comportamiento reimplementado el método especia *__eq__()*.

__eq__()

```
def __eq__(self, otro):
 return self.x == otro.x and self.y == otro.y
```

Python provee también el operador de desigualdad (!=) *__neq__()* (not equal) automáticamente si se implementa el *__eq__()* pero no se implementa *__neq__()*.

Con la implementación de métodos especiales se pueden comparar objetos de tipo *Punto* pero si se va a comparar un *Punto* con un objeto de diferente tipo, por ejemplo *int*, se producirá una excepción *AtrributeError* porque los *ints* no tienen un atributo *x*.

Por otra parte, se pueden comparar objetos Puntos con otros objetos que tengan los mismos atributos debido al tipado del pato de Python[17]. No

[17]Este tipo de operaciones no es recomendable utilizar dado que se pueden comparar

obstante, este tipo de comparaciones pueden conducir a resultados sorprendentes.

Si se desean evitar comparaciones inapropiadas existen unos pocos enfoques que se pueden aplicar. Uno de ellos es usar aserciones por ejemplo *assert isinstance(otro,Punto)*. Otra es disparar una excepción *TypeError* es decir *if not isinstance(otro,Punto): raise TypeError()*. La tercera forma y quizás la más pitónica es: *if not isinstance(otro,Punto): return NotImplemented*. En este último caso, si se retorna *NotImplemented*, Python intentará llamar a otro *__eq__()* si el otro tipo soporta una comparación con el tipo Punto y si no existe tal método o si el método también retorna *NotImplemented*, Python disparará una excepción *TypeError*[18]. La función primitiva *isinstance()* toma como argumento un objeto y una clase (o una tupla de clases) y retorna True si el objeto es de la clase recibida como parámetro (o de una de las tuplas de clases) o de una de las clases bases (considere las clases de la tupla de clases).

El método primitivo *repr()* invoca al método especial *__repr__()* con el objeto receptor y retorna una cadena de caracteres como resultado.

__repr__()

```
def __repr__(self):
 return "Punto({0.x!r}, {0.y!r})".format(self)
```

La cadena de caracteres retornada puede ser de dos clases. La primera de ellas, es una cadena que puede ser evaluada usando la primitiva *eval()* la cual produce un objeto equivalente al que invocó a *repr()*. La otra se usa donde esto no es posible se verá un ejemplo más adelante.

objetos que pertenecen a clases diferentes para los cuales la comparación no tenga sentido.

[18]Note que solo las reimplementaciones de los métodos especiales de comparación pueden retornar *NotImplemented*

repr

```
p = Forma.Punto(3, 9)
repr(p) # retorna: 'Punto(3, 9)'
q = eval(p.__module__ + "." + repr(p))
repr(q) # retorna: 'Point(3, 9)'
```

Al final de este trozo de código se tienen dos puntos p y q con los mismos atributos y por lo tanto se pueden comparar por igual. La función *eval()* retorna el resultado de ejecutar la cadena de caracteres que recibe como parámetro, por lo tanto, la misma debe contener sentencias de Python válidas.

__str__

```
def __str__(self):
 return "({0.x!r}, {0.y!r})".format(self)
```

La función primitiva *__str__()* se comporta de la misma forma que *repr()* excepto que llama al método especial *__str__()*. Se espera que el resultado de este método sea legible por los programadores y no pasado como parámetro a *eval()*. Si se ejecuta *str(p)* y *str(q)* se retorna la cadena *'(3,9)'*.

6.4.2. Herencia y Polimorfismo

La clase *Círculo* se construye a partir de la clase *Punto* usando herencia. La clase *Círculo* incorpora un dato adicional, el *radio*, y tres métodos nuevos. También reimplementa algunos pocos métodos de *Punto*. A continuación se muestra una implementación de Círculo.

Círculo

```
class Círculo(Punto):
 def __init__(self, radio, x=0, y=0):
  super().__init__(x, y)
  self.radio = radio

 def distanciaAlBordeDesdeElOrigien(self):
  return abs(self.distanciaDesdeElOrigen() -
             self.radio)

 def área(self):
  return math.pi * (self.radio ** 2)

 def circunsferencia(self):
  return 2 * math.pi * self.radio

 def __eq__(self, otro):
  return self.radio == otro.radio and super()
                       .__eq__(otro)

 def __repr__(self):
  return "Círculo({0.radio!r},␣{0.x!r},␣{0.y!r})"
         .format(self)

 def __str__(self):
  return repr(self)
```

La herencia se realiza simplemente listando la clase (o las clases *En el caso de la Herencia Múltiple*) que se desean que la clase en desarrollo herede. En este caso se ha heredado solamente de la clase *Punto*.

Dentro del método *__init__()* se usa el *super()* con el propósito de llamar al método *__init__()* de la clase base el cual crea los atributos

self.x y *self.y*. Los usuarios de la clase podrían proveer un radio inválido, tal como -2, más adelante se verá como prevenir tales tipos de problemas a través del uso de propiedades. Los métodos *área()* y *circunsferencia()* son directos. El método *distanciaAlBordeDesdeElOrigen()* llama al método *distanciaDesdeElOrigen()* como parte de su computación. Dado que *Círculo* no ha provisto una implementación de *distanciaDesdeElOrigen()* se utiliza la implementación heredada desde *Punto*. Esto contrasta con la reimplementación del método *__eq__()*. Este método compara este radio del círculo con el otro radio del círculo y si ellos son iguales entonces explícitamente llama al método *__eq__()* de la clase base usando *super()*. En el caso de no usar *super* se producirá una recursión infinita porque *Círculo.__eq__()* se invocaría a sí mismo. Note que no se tienen que pasar los argumentos a *super()* dado que Python los pasa automáticamente.

Ejemplos de uso de Círculo

```
p = Forma.Punto(28, 45)
c = Forma.Círculo(5, 28, 45)
p.distanciaDesdeElOrigen() # returna: 53.0
c.distanciaDesdeElOrigen() # returna: 53.0
```

Se puede llamar el método *distanciaDesdeElOrigen()* para un *Punto* o para un *Círculo* dado que los círculos pueden sustituir a los puntos.

Polimorfismo significa que cualquier objeto de una clase dada puede ser usado como si fuese un objeto de alguna de sus clases bases [19]. Esto se debe a que cuando se crea una subclase se necesitan implementar solo los métodos adicionales y se tienen que reimplementar los métodos que se desean reemplazar, y cuando se reimplementan esos métodos, se pueden usar los métodos de las clases bases usando *super()* dentro de la redefinición.

En el caso de la clase *Círculo* se han implementado métodos adicionales tales como *área()* y *circunsferencia()*, y reimplementado los métodos que se necesitan cambiar. Las redefiniciones de *__repr__()* y *__str__()* son ne-

[19] Esto se conoce como principio de sustitución.

cesarias porque sin ellas se utilizan los métodos de la clase base y retornan cadenas que representan puntos en vez de las cadenas que representan círculos. Las redefiniciones de *__init__()* y *__eq__()* son necesarias porque se debe tener en cuenta que los círculos tienen un atributo adicional, y en ambos casos se hace uso de la implementación de la clase base.

6.4.3. Ejercicios

Ejercicio 1: Defina una clase con una variable de instancia privada. Luego muestre que la privacidad es llevada a cabo por convención.

Ejercicio 2: Defina la clase Persona. La clase permite registrar el nombre y la edad de una persona. Luego escriba un programa principal que permita que el usuario ingrese n personas a una lista. Finalmente, el programa debe imprimir la lista por pantalla.

Ejercicio 3: Defina la clase EmpleadoPorHora la clase permite almacenar el nombre del empleado, el nombre del lugar donde trabaja, el precio por hora y la cantidad de horas que trabaja.

Ejercicio 4: Defina la clase EmpleadoExclusivo. La clase permite registrar el nombre del empleado, el lugar donde trabaja y el sueldo que cobra por mes.

Ejercicio 5: Escriba un programa que permita que el usuario ingrese EmpleadosPorHora y EmpleadosSemiExclusivos en una lista. Luego imprima por pantalla el empleado por hora que gana más dinero y el empleado semiexclusivo que tiene sueldo más alto.

Ejercicio 6: A través del análisis de las clases definidas en los ejercicio 3 y 4 y en programa creado en 5 elabore una solución con la utilización de herencia. Luego compare las soluciones.

Ejercicio 7: Realice las siguientes actividades:

1. Defina la clase Animal la cual permite almacenar el grupo al cual pertenece el animal: Vertebrado, Invertebrado, Anélidos, Moluscos, Poríferos, Cnidarios, Nematodos, Platelmintos .
2. Defina la clase Perro la cual permite almacenar la raza de un perro.
3. Defina la clase Pez la cual permite almacenar si es de agua dulce o salada.
4. Escriba un programa principal que permita:
   - Almacenar en una lista perros y peces.
   - Imprimir la lista.

Ejercicio 8: Defina la clase ContadorLimitado. Un contador limitado es un contador que puede contar hasta un límite establecido por el programador. La clase soporta las siguientes operaciones:

1. Inicializar el contador en algún valor menor o igual que el valor máximo admitido.
2. Retornar el valor actual del contador.
3. Incrementar el contador en uno.
4. Devolver una representación del contador admitida por eval().

Ejercicio 9: Crea una jerarquía de clases que permita organizar los siguientes conceptos: *Los animales se dividen en varios subgrupos, algunos de los cuales son vertebrados: (aves, mamíferos, anfibios, reptiles, peces) e invertebrados: artrópodos (insectos, arácnidos, miriápodos, crustáceos), anélidos (lombrices, sanguijuelas), moluscos (bivalvos, gasterópodos, cefalópodos), poríferos (esponjas), cnidarios (medusas, pólipos, corales), equinodermos (estrellas de mar), nematodos (gusanos cilíndricos), platelmintos (gusanos planos)*

Ejercicio 10: Realice las siguientes actividades:

1. Defina la clase Empresa la cual permite registrar el nombre de la empresa, dirección y cuit.

2. Defina la clase RenglónDeFactura. La clase registra la siguiente información:
   a) Cantidad de unidades de una mercadería.
   b) La descripción de la mercadería.
   c) El precio unitario la mercadería.
   d) El precio total (todas las unidades de la mercadería).
3. Defina la clase Factura, la que permite registrar el número de la factura, la empresa, las mercaderías que compran, también permite calcular el total sin un porcentaje de IVA, el porcentaje de IVA y el total con IVA.
4. Escriba un programa principal que permita que:
   a) El usuario ingrese n facturas y las almacene en una diccionario de facturas. El número de la factura es la clave y la factura en si es el valor.
   b) Imprima el nombre del cliente que hizo la mayor compra.
   c) Imprima el total de ventas.
   d) Imprima todos los números de facturas.

6.4.4. Uso de Propiedades para Contorlar el Acceso a los Atributos

Con *property()* Python, se pueden crear atributos administrados en las clases. Este tipo de atributos, también conocidos como propiedades, se pueden utilizar cuando se necesita modificar la implementación interna sin cambiar la API[20] pública de la clase. Proporcionar APIs estables puede ayudar a evitar romper el código de los usuarios que usan las clases y objetos.

Podría decirse que las propiedades es la forma más popular de crear atributos administrados rápidamente y al más puro estilo pitónico.

[20] Application Program Interface.

Administración de Atributos en las Clases

Cuando se define una clase en un lenguaje de programación orientado a objetos, probablemente se necesitarán atributos de clase e instancia. En otras palabras, se requerirán variables a las que se puede acceder a través de la instancia, la clase o incluso ambas, dependiendo del lenguaje. Los atributos mantienen el estado interno de un objeto determinado, al que a menudo se necesitará acceder y mutar.

Por lo general, se tienen dos formas de administrar un atributo. Se puede acceder y mutar el atributo directamente o se pueden usar métodos. Los métodos son funciones vinculadas a una clase dada y proporcionan los comportamientos y acciones que un objeto puede realizar con sus datos y atributos internos.

Si se exponen los atributos al usuario, se vuelven parte de la API pública de la clase. El usuario accederá y los mutará directamente en su código. El problema surge cuando se necesita cambiar la implementación interna de un atributo.

Suponga que se está trabajando con la clase Círculo. La implementación inicial tiene un solo atributo llamado radio (más los heredados de la clase Punto). Una vez finalizada la codificación de la clase se la pone a disposición de sus usuarios finales. Dichos usuarios comienzan a usar Círculo en su código para crear muchos proyectos y aplicaciones.

Ahora suponga que tiene un usuario importante que requiere un cambio: No necesita que Círculo almacene el radio sino el diámetro.

En este punto, eliminar radio y comenzar a usar diámetro provoca que el código de algunos de usuarios deje de funcionar. Se debe manejar esta situación de otra manera sin eliminar radio.

Lenguajes de programación como Java y C++ alientan a no exponer los atributos para evitar este tipo de problema. En su lugar, se deben proporcionar métodos *getter* y *setter*, también conocidos como *observadores* y *modificadores*, respectivamente. Estos métodos ofrecen una forma de cambiar la implementación interna de sus atributos sin cambiar su API pública.

La Aproximación con Getters y Setters en Python

Técnicamente, no hay nada que impida usar métodos *getter* y *setter* en Python. A continuación se muestra este enfoque.

Punto con Observadores y Modificadores

```
class Punto:
 def __init__(self, x, y):
  self._x = x
  self._y = y

 def get_x(self):
  return self._x

 def set_x(self, valor):
  self._x = valor

 def get_y(self):
  return self._y

 def set_y(self, valor):
  self._y = valor
```

En este ejemplo, se crea *Punto* con dos atributos no públicos $._x$ y $._y$ los cuales contienen las coordenadas cartesianas del punto en cuestión.

Comentario Importante

Python no tiene la noción de modificadores de acceso, tales como: privado, protegido y público, para restringir el acceso a atributos y métodos. En Python, la distinción es entre miembros de clase públicos y no públicos.

Si se desea indicar que un atributo o método dado no sea público, se debe usar la conocida convención de Python de anteponer al nombre un guión bajo (_). Esa es la razón detrás de la denominación de los atributos $._x$ y $._y$.

Tenga en cuenta que esto es solo una convención. No impide que ud. y otros programadores accedan a los atributos mediante la notación de puntos, como en *obj._attr*. Sin embargo, es una mala práctica violar esta convención.

Para acceder y mutar el valor de $._x$ o $._y$, puede usar los métodos getter y setter correspondientes.

Uso de Punto con Getters y Setters

```
>>> from punto import Punto

>>> punto = Punto(12, 5)
>>> punto.get_x()
12
>>> punto.get_y()
5

>>> punto.set_x(42)
>>> punto.get_x()
42

>>> # Los atributos no públicos son accesibles
>>> punto._x
42
>>> punto._y
5
```

Con *.get_x()* y *.get_y()*, puede acceder a los valores actuales de *._x* y *._y*. Se puede usar el método setter para almacenar un nuevo valor en el atributo administrado correspondiente. A partir de este código, se puede confirmar que Python no restringe el acceso a atributos no públicos.

El Enfoque Pitónico

A continuación se muestra el ejemplo previo escrito de una manera concisa y más pitónica.

Enfoque Pitónico

```
>>> class Punto:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...

>>> punto = Punto(12, 5)
>>> punto.x
12
>>> punto.y
5

>>> punto.x = 42
>>> punto.x
42
```

Este código descubre un principio fundamental. Exponer atributos al usuario final es normal y común en Python [21]. No se necesita implementar en las clases los métodos getter y setter todo el tiempo. Sin embargo, ¿cómo se pueden manejar los cambios en los requisitos que implican cambios en la API?

A diferencia de Java y C++, Python proporciona herramientas útiles que permiten cambiar la implementación subyacente de los atributos sin modificar su API pública. El enfoque más popular es convertir sus atributos en *propiedades*.

Las propiedades representan una funcionalidad intermedia entre un atributo (o campo) y un método. En otras palabras, permiten crear métodos que se comportan como atributos. Con las propiedades, se puede cambiar la forma en se que calcula el atributo siempre que se necesite.

[21] Recuerde que este tipo de prácticas viola la encapsulación la cual es una propiedad fundamental en la Programación Orientada a Objetos

Por ejemplo, se puede convertir tanto *.x* como *.y* en propiedades. Con este cambio, se puede continuar accediendo a ellos como atributos. También se tendrá un método subyacente que contiene *.x* y *.y* que permitirá modificar su implementación interna y realizar acciones con ellos antes de que los usuarios accedan y los transformen. La principal ventaja de las propiedades de Python es que le permiten exponer sus atributos como parte de su API pública.

Propiedades con Python

property() de Python es la forma pitónica de evitar los métodos getters y setters en el código. Esta función permite convertir atributos de la clase en *propiedades* o *atributos administrados*. Dado que *property()* es una función integrada, se puede usar sin inconvenientes.

Con *property()*, se pueden vincular métodos getter y setter a los atributos de la clase. De esta forma, es posible manejar la implementación interna de ese atributo sin los métodos getter y setter en su API. También puede especificar una forma de manejar la eliminación de atributos y proporcionar una cadena de documentación adecuada para sus propiedades. A continuación se muestra la sintaxis de *property()*

Signatura de property

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Los dos primeros argumentos son funciones que desempeñarán el papel de los métodos getter (fget) y setter (fset). Los siguientes ítems describen cada argumento:

fget: función que retorna el valor de un atributo.

fset: función que permite cambiar el valor de un atributo.

fdel: función que define como un atributo maneja la supresión.

doc: un docstring del atributo.

El valor de retorno de *property()* es el propio atributo gestionado. Si accede al atributo, como en *obj.attr*, Python llama automáticamente a *fget()*. Si asigna un nuevo valor al atributo, como en *obj.attr = valor*, Python llama a *fset()* usando el valor de entrada como argumento. Finalmente, si ejecuta una instrucción *del obj.attr*, Python llama automáticamente a *fdel()*.

Se puede usar *doc* para proporcionar una cadena de documentación adecuada para las propiedades. Será posible leer esa cadena de documentación usando la ayuda de Python *help()*. El argumento *doc* también es útil cuando trabaja con editores de código e IDEs que admiten el acceso a cadenas de documentos.

Se puede usar *property()* ya sea como una función o un decorador para implementar propiedades. En las dos secciones siguientes, aprenderá a utilizar ambos enfoques. Sin embargo, se debe saber que el enfoque del decorador es más popular en la comunidad de Python.

Creación de Atributos con property()

Se puede crear una propiedad llamando a *property()* con un conjunto apropiado de argumentos y asignando su valor devuelto a un atributo de clase.

El siguiente ejemplo muestra cómo crear una clase *Círculo* con una propiedad útil para administrar su radio:

Círculo - Propiedades

```
class Círculo:
 def __init__(self, radio):
  self._radio = radio

 def _get_radio(self):
  print("Get radio")
  return self._radio
```


Círculo - Propiedades - Continuación

```
def _set_radio(self, valor):
 print("Set radio")
 self._radio = valor

def _del_radio(self):
 print("Delete radio")
 del self._radio

radio = property(
 fget=_get_radio,
 fset=_set_radio,
 fdel=_del_radio,
 doc="La propiedad radio."
)
```

En este fragmento de código, se crea la clase *Círculo*. El inicializador de clase .__*init*__(), toma el radio como argumento y lo almacena en un atributo no público llamado ._*radio*. Luego define tres métodos no públicos:

1. ._get_radio() devuelve el valor actual de ._*radio*.
2. ._set_radio() toma valor como argumento y lo asigna a ._*radio*.
3. ._del_radio() elimina el atributo de instancia ._*radio*.

Una vez que tenga estos tres métodos en su lugar, se puede crear una instancia de *Círculo* y luego acceder al atributo *radio*.

Uso de Círculo con Propiedades

```
>>> from círculo import Círculo

>>> círculo = Círculo(42.0)

>>> círculo.radio
Get radio
42.0

>>> círculo.radio = 100.0
Set radio
>>> círculo.radio
Get radio
100.0
>>> del círculo.radio
Delete radio
```

La propiedad *.radio* oculta la variable de instancia privada *._radio*, que en este ejemplo es un atributo administrado. Se puede acceder y asignar *.radio* directamente. Internamente, Python llama automáticamente a *._get_radio()* y *._set_radio()* cuando sea necesario. Cuando se ejecuta *del circulo.radio*, Python llama a *._del_radio()*, que elimina el *._radio* subyacente.

Las propiedades son atributos de clase que administran atributos de instancia. Se puede pensar en una propiedad como una colección de métodos agrupados. Si inspecciona *.radio* cuidadosamente, es posible encontrar los métodos que se proporcionaron como argumentos a *fget*,*fset* y *fdel*.

Información de Propiedad

```
>>> from círculo import Círculo

>>> Círculo.radio.fget
<function Circulo._get_radio at 0x7fba7e1d7d30>

>>> Círculo.radio.fset
<function Circulo._set_radio at 0x7fba7e1d78b0>

>>> Círculo.radio.fdel
<function Círculo._del_radio at 0x7fba7e1d7040>

>>> dir(Círculo.radio)
[..., '__get__', ..., '__set__', ...]
```

Uso de property() como Decorador

Los *decoradores* están en todas partes en Python. Son funciones que toman otra función como argumento y devuelven una nueva función con funcionalidad añadida. Con un decorador, se puede adjuntar operaciones de procesamiento previo y posterior a una función existente.

Cuando Python 2.2 introdujo *property()*, la sintaxis del decorador no estaba disponible. La única forma de definir propiedades era pasar los métodos *getter*, *setter* y *deleter*, como se explicó previamente. La sintaxis del *decorador* se agregó en Python 2.4 y, en la actualidad, usar *property()* como decorador es la práctica más popular en la comunidad de Python.

La sintaxis del decorador consiste en colocar el nombre de la función decoradora con un símbolo @ inicial justo antes de la definición de la función que desea decorar:

Sintaxis de un Decorador

```
@decorador
def func(a):
 return a
```

En este fragmento de código, *@decorador* puede ser una función o clase destinada a decorar *func()*. Esta sintaxis es equivalente a la siguiente:

Sintaxis Equivalente de un Decorador

```
def func(a):
 return a

func = decorador(func)
```

La última línea de código hace *func* contenga el resultado de llamar a *decorador(func)*. Tenga en cuenta que esta es la misma sintaxis que usó para crear una propiedad en la sección anterior.

property() de Python también puede funcionar como decorador, por lo que puede usar la sintaxis *@property* para crear sus propiedades rápidamente:

Círculo con la Propiedad como Decorador

```
1 class Círculo:
2
3  def __init__(self, radio):
4   self._radio = radio
5
6  @property
7  def radio(self):
8   """La Propiedad Radio."""
9   print("Get radio")
10  return self._radio
11
12 @radio.setter
13 def radius(self, valor):
14  print("Set radio")
15  self._radio = valor
16
17 @radio.deleter
18 def radio(self):
19  print("Delete radio")
20  del self._radio
```

Este código se ve bastante diferente del enfoque de los métodos *getter* y *setter*. *Círculo* ahora se ve más pitónico y limpio. Ya no se necesita usar nombres de métodos como *._get_radio()*, *._set_radio()* y *._del_radio()*. Tiene tres métodos con el mismo nombre claro y descriptivo similar a un atributo. El enfoque del decorador para crear propiedades requiere definir un primer método utilizando el nombre público del atributo administrado subyacente, que en este caso es *.radio*. Este método debería implementar la lógica *getter*. En el ejemplo anterior, las líneas 7 a 10 implementan ese método.

Las líneas 13 a 15 definen el método *setter* para *.radio*. En este caso, la

sintaxis es bastante diferente. En lugar de usar *@property* nuevamente, usa *@radio.setter*.

Además de *.fget*, *.fset*, *.fdel* y muchos otros atributos y métodos especiales, la propiedad también proporciona *.deleter()*, *.getter()* y *.setter()*. Cada uno de estos tres métodos devuelve una nueva propiedad.

Cuando se decora el segundo método *.radio()* con *@radio.setter* (línea 12), se crea una nueva propiedad y se reasigna el nombre de nivel de clase *.radio* (línea 15) para contenerla. Esta nueva propiedad contiene el mismo conjunto de métodos de la propiedad inicial con adición del nuevo método *setter* proporcionado en la línea 13. Finalmente, la sintaxis del decorador reasigna la nueva propiedad al nombre de nivel de clase *.radio*.

El mecanismo para definir el método de eliminación es similar. Esta vez, necesita usar el decorador *@radio.deleter*. Al final del proceso, se obtiene una propiedad completa con los métodos *getter*, *setter* y *deleter*.

Finalmente, ¿cómo se pueden proporcionar docstrings adecuadas para sus propiedades cuando utiliza el enfoque de decorador? Se notará que ya se hizo al agregar un docstring al método *getter* en la línea 8.

La nueva implementación de *Círculo* funciona igual que el ejemplo de la sección anterior:

Funcionamiento de Círculo con Propiedades

```
>>> from círculo import Círculo

>>> círculo = Círculo(42.0)

>>> círculo.radio
Get radio
42.0
```


Funcionamiento de Círculo con Propiedades

```
>>> círculo.radio = 100.0
Set radio
>>> círculo.radio
Get radio
100.0

>>> del círculo.radio
Delete radio
>>> círculo.radio
Get radio
Traceback (most recent call last):
...
AttributeError: 'Círculo' object has no attribute
 '_radio'

>>> help(círculo)
Help on Círculo in module __main__ object:

class Círculo(builtins.object)
...
|  radio
|      The radio property.
```

No se necesita usar un par de paréntesis para llamar a *.radio()* como método. En su lugar, se puede acceder a *.radio* como accedería a un atributo normal, que es el uso principal de las propiedades. Las propiedades permiten tratar los métodos como atributos y se encargan de llamar automáticamente al conjunto subyacente de métodos.

A continuación se presenta un resumen de algunos puntos importantes para recordar cuando se esté creando propiedades con el enfoque de decorador:

- El decorador *@property* debe decorar el método *getter*.
- La cadena de documentación debe ir en el método *getter*.
- Los métodos *setter* y *deleter* deben estar decorados con el nombre del método *getter* más *.setter* y *.deleter*, respectivamente.

Comentario Importante

En general, se debe evitar convertir atributos que no requieren procesamiento adicional en propiedades. El uso de propiedades en esas situaciones puede hacer el código:

- Innecesariamente detallado
- Confundir a otros desarrolladores
- Más lento que el código basado en atributos regulares

A menos que se necesite algo más que el acceso a atributos básicos, no se deben escribir propiedades. Son una pérdida de tiempo de la CPU y, lo que es más importante, es una pérdida de tiempo para el programador. Finalmente, se debe evitar escribir métodos *getter* y *setter* explícitos y luego envolverlos en una propiedad. En su lugar, utilice el decorador *@property*. Esa es la forma más pitónica de hacerlo.

Atributos de solo Lectura

Probablemente el caso de uso más elemental de *property()* es proporcionar atributos de solo lectura en sus clases. Si se necesita una clase *Punto* inmutable que no permita al usuario cambiar el valor original de sus coordenadas, x e y se puede hacer lo siguiente:

Atributos de Solo Lectura

```
class Punto:
 def __init__(self, x, y):
  self._x = x
  self._y = y

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

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

Aquí, se almacena los argumentos de entrada en los atributos *._x* y *._y*. Como ya se aprendió, el uso del guión bajo (_) en los nombres indica a otros desarrolladores que son atributos no públicos y que no se debe acceder a ellos mediante la notación de puntos, como en *punto._x*. Finalmente, define dos métodos *getter* y los decora con *@property*.

Ahora se tiene dos propiedades de solo lectura, .x y .y, como sus coordenadas:

Atributos de Solo Lectura

```
>>> from punto import Punto
>>> punto = Punto(12, 5)
>>> # Leer cordenadas
>>> punto.x
12
```


Atributos de Solo Lectura

```
>>> punto.y
5
>>> # Escribir coordenadas
>>> punto.x = 42
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

Aquí, *punto.x* y *punto.y* son ejemplos básicos de propiedades de solo lectura. Su comportamiento se basa en el descriptor subyacente que proporciona la propiedad. Como se observa, la implementación predeterminada de *.__set__()* genera un *AttributeError* cuando no define un método de iniciación adecuado.

Se puede llevar esta implementación de *Punto* un poco más lejos y proporcionar métodos setters explícitos que generan una excepción personalizada con mensajes más elaborados y específicos:

Atributos de Solo Lectura - Setters Explícitos

```
class ErrorDeEscrituraDeCoordenadas(Exception):
 pass

class Punto:
 def __init__(self, x, y):
 self._x = x
 self._y = y

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


Atributos de Solo Lectura - Setters Explícitos

```
class Punto
 ...
 @x.setter
 def x(self, value):
  raise ErrorDeEscrituraDeCoordenadas("x es de solo\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣lectura")

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

 @y.setter
 def y(self, valor):
  raise ErrorDeEscrituraDeCoordenadas("y es de solo\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣lectura")
```

En este ejemplo, define una excepción personalizada llamada *ErrorDeEscrituraDeCoordenadas*. Esta excepción le permite personalizar la forma en que implementa su clase *Punto* inmutable. Ahora, ambos métodos setters generan su excepción personalizada con un mensaje más explícito.

Atributos de Lectura y Escritura

También se puede usar *property()* para proporcionar atributos administrados con capacidades de lectura y escritura. En la práctica, solo se necesita proporcionar el método getter ("leer") y el método setter ("escribir") apropiados a las propiedades para crear atributos administrados de lectura y escritura.

Suponga que se desea que la clase *Círculo* tenga un atributo *.diámetro*. Sin embargo, usar el *radio* y el *diámetro* en el *inicializador de clase* parece innecesario porque puede calcular uno usando el otro. A continuación

se muestra un círculo que administra .radio y .diámetro como atributos de lectura y escritura:

Atributos de Solo Lectura y Escritura

```
import math

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

 @property
 def radio(self):
  return self._radio

 @radio.setter
 def radio(self, valor):
  self._radio = float(valor)

 @property
 def diámetro(self):
  return self.radio * 2

 @diámetro.setter
 def diámetro(self, valor):
  self.radio = valor / 2
```

En el código, se crea una clase *Círculo* con un radio de lectura y escritura. En este caso, el método getter solo devuelve el valor del radio. El método setter convierte el valor de entrada y se lo asigna a *._radio* no público, que es la variable que usa para almacenar los datos finales.

Hay un detalle sutil a tener en cuenta en esta nueva implementación de Círculo y su atributo *.radio*. En este caso, el inicializador de la clase asigna el valor de entrada a la propiedad *.radio* directamente en lugar de almacenarlo

en un atributo no público dedicado, como *._radio*.

¿Por qué? Porque debe asegurarse de que cada valor proporcionado como un radio, incluido el valor de inicialización, pase por el método setter y se convierta en un número de coma flotante.

La clase *Círculo* también implementa un atributo *.diámetro* como propiedad. El método getter calcula el diámetro usando el radio. El método setter hace algo curioso. En lugar de almacenar el valor del diámetro de entrada en un atributo dedicado, calcula el radio y escribe el resultado en *.radio*.

Funcionamiento de la Nueva Versión de Círculo

```
>>> from círculo import Círculo

>>> círculo = Círculo(42)
>>> círculo.radio
42.0

>>> círculo.diámetro
84.0

>>> círculo.diámetro = 100
>>> círculo.diámetro
100.0

>>> círculo.radio
50.0
```

Atributos de Solo Escritura

También se puede crear atributos de solo escritura modificando la forma en que se implementa el método getter de las propiedades. Por ejemplo, puede hacer que el método getter genere una excepción cada vez que un usuario acceda al valor del atributo subyacente.

Aquí hay un ejemplo de manejo de contraseñas con una propiedad de solo escritura:

Atributo de Solo Escritura

```
import hashlib
import os

class Usuario:
 def __init__(self, nombre, clave):
  self.nombre = nombre
  self.clave  = clave
  self._hashed_clave=""

 @property
 def clave(self):
  raise AttributeError("clave es de solo escritura")

 @clave.setter
 def clave(self, texto):
  salt = os.urandom(32)
  self._hashed_password = hashlib.pbkdf2_hmac(
    "sha256", texto.encode("utf-8"), salt, 100_000
  )
```

El inicializador de Usuario toma un nombre de usuario y una contraseña como argumentos y los almacena en *.nombre* y *.clave*, respectivamente. Utiliza una propiedad para administrar el procesamiento de la contraseña de entrada. El método getter genera un *AttributeError* cada vez que un usuario intenta recuperar la contraseña actual. Esto convierte *.clave* en un atributo de solo escritura:

Atributo de Solo Escritura

```
>>> from usuarios import Usuario

>>> juan = User("Juan", "secreto")

>>> juan._hashed_clave
b'b\xc7^ai\x9f3\xd2g␣...␣\x89^—\x92\xbe\xe6'

>>> john.clave
Traceback (most recent call last):
...
AttributeError: Clave es de solo lectura

>>> john.calve = "super-secreto"
>>> john._hashed_clave
b'\xe91$\x9f\xaf\x9d␣...␣b\xe8\xc8\xfcaU\r_'
```

En este ejemplo, crea a *juan* como una instancia de usuario con una contraseña inicial. El método setter codifica la contraseña y la almacena en *._hashed_clave*. Se tiene que tener en cuenta que cuando se intenta acceder a *.clave* directamente, se obtiene un *AttributeError*.

Finalmente, la asignación de un nuevo valor a *.clave* activa el método setter y crea una nueva contraseña cifrada.

En el método setter de *.clave os.urandom()* genera una cadena aleatoria de 32 bytes como salida de su función de *hash*. Para generar la contraseña cifrada, se utiliza *hashlib.pbkdf2_hmac()*. Luego, se almacena la contraseña resultante en el atributo no público *._hashed_password*. Si se hace de esta manera, se asegurará de que nunca se guarde la contraseña de texto sin formato en ningún atributo recuperable.

Comentario

Una propiedad es un tipo especial de miembro de clase que proporciona una funcionalidad que se encuentra en algún lugar entre los atributos y métodos regulares. Las propiedades le permiten modificar la implementación de los atributos de instancia sin cambiar la API pública de la clase. Es posible mantener las APIs sin cambios lo que evita romper el código que los usuarios escribieron sobre versiones anteriores de las clases.

Las propiedades son la forma pitónica de crear atributos administrados en las clases. Existen varios casos de uso en la programación del mundo real, lo que las convierte en una gran incorporación a su conjunto de habilidades como desarrollador de Python.

Ejercicios

Ejercicio 1: Defina la clase Persona la cual permite almacenar el nombre y la edad de una persona. Resuelva este ejercicio utilizando propiedades.

Ejercicio 2: Defina la clase Botón que permite almacenar el estado (Presionado, Libre) de un botón. Resuelva este ejercicio utilizando propiedades.

Ejercicio 3: Defina la clase semáforo la cual tiene tres variables: rojo, verde, amarillo. Las variables se pueden inicializar con un método privado de la clase y para el usuario son de solo lectura.

Ejercicio 4: Defina la clase SemaforoConBotón la cual tiene un botón y un semáforo. El semáforo funciona de la siguiente manera: Se presiona el botón se enciende la luz roja, se presiona nuevamente se apaga la luz roja, cuando se presiona de nuevo se prende la luz verde, si se presiona nuevamente se apaga la luz verde, se presiona nuevamente se prende la luz amarilla y luego de otra presión se apaga la luz amarilla y comienza el ciclo nuevamente.

Ejercicio 5: Defina la clase Password la cual tiene un atributo de solo escritura privado. Cuando un usuario desea leer el atributo se devuelve un error de lectura.

Ejercicio 6: Defina la clase MatrizCuadrada la cual permite almacenar números flotantes. La clase permite realizar las siguientes operaciones:

1. Cargar una matriz.
2. Asignar a una variable una matriz.
3. Sumar dos matrices.
4. Restar dos matrices.
5. Construir una representación textual de una matriz.

Nota: Para resolver este ejercicio sobrecargue los operadores que crea conveniente.

Ejercicio 7: Defina la clase ListaDeControlyEstado. Este tipo de lista tiene tres posiciones con las siguientes funcionalidades:

1. Las posiciones cero y uno son de lectura y escritura.
2. La posición 7 es de solo lectura y se coloca en 1 cuando las posiciones cero y uno están con el valor 1 en otro caso tiene el valor 0.

Ejercicio 8: Defina la clase ALU la cual tiene dos variables que permiten almacenar operandos numéricos y una variable que permite almacenar un resultado el cual es de solo lectura y se inicializa con el resultado de realizar una operación con los operandos. Las operaciones que la ALU puede realizar son: suma, resta, multiplicación y división.

Ejercicio 9: Dado el siguiente programa:

```
class Powers:
 def __init__(self, square, cube):
  self._square = square    # _square is the base value
```


```
        self._cube   = cube      # square is the property name

    def getSquare(self):
        return self._square ** 2

    def setSquare(self, value):
        self._square = value

    square = property(getSquare, setSquare)

    def getCube(self):
        return self._cube ** 3

    cube = property(getCube)


X = Powers(3, 4)
print(X.square)       # 3 ** 2 = 9
print(X.cube)         # 4 ** 3 = 64
X.square = 5
print(X.square)       # 5 ** 2 = 25
```

Se pide:

1. Diga si el programa usa propiedades.
2. Qué hace el programa?

6.4.5. Creación de Tipos de Datos Integrados

A la hora de crear un tipo de datos completo se tienen dos posibilidades. Una es crear el tipo de datos desde cero. Aunque el tipo de datos heredará de *object* (como lo hacen todas las clases de Python), se deben proporcionar todos los atributos y métodos que requiere el tipo de datos (aparte de *__new__()*). La otra posibilidad es heredar de un tipo de datos existente

que sea similar al que se desea crear. En este caso, el trabajo suele implicar volver a implementar aquellos métodos que se comportan de manera diferente y *ocultar* aquellos métodos que no son requeridos.

En la subsección siguiente se implementará el tipo de datos *BooleanoDifuso* desde cero, y en la subsección siguiente se implementará el mismo tipo pero se usará herencia para reducir el trabajo. El tipo booleano incorporado tiene dos valores (*True* y *False*), pero en algunas áreas de IA (Inteligencia Artificial) se utilizan booleanos difusos, que tienen valores correspondientes a *True* y *False*, y también a intermedios entre ellos. En las implementaciones se usarán valores de punto flotante con 0.0 para denotar False y 1.0 para indicar True. En este sistema, 0,5 significa 50 por ciento de verdad, y 0,25 significa 25 por ciento de verdad, y así sucesivamente. A continuación se muestran algunos ejemplos de uso (funcionan igual con cualquier implementación):

Atributo de Solo Escritura

```
a = BoolenoDifuso.BoolenoDifuso(.875)
b = BoolenoDifuso.BoolenoDifuso(.25)
a >= b # retorna: True
bool(a), bool(b) # retorna: (True, False)
~a # retorna: BoolenoDifuso(0.125)
a & b # retorna: BoolenoDifuso(0.25)
b |= BoolenoDifuso.BoolenoDifuso(.5)
# b es ahora: BoolenoDifuso(0.5)
"a={0:.1%} b={1:.0%}".format(a, b)
# retorna: 'a=87.5% b=50%'
```

Se desea que el tipo *BooleanoDifuso* admita el conjunto completo de operadores de comparación (<, <=, ==, !=, >=, >) y las tres operaciones lógicas básicas, not (~), and (&), y or (|). Además de las operaciones lógicas, se pretende proporcionar un par de otros métodos lógicos, *conjunción()* y *disyunción()*, que toman tantos *BooleanosDifusos* como se desee y devuelven el *BooleanoDifuso* resultante apropiado. Para completar el tipo de datos, se quiere proporcionar conversiones a tipos *bool*, *int*, *float*, y *str*, y se desea te-

ner una forma de representación compatible con *eval()*. Los requisitos finales son que *BooleanoDifuso* admita las especificaciones de formato *str.format()*, que los *BooleanosDifusos* se puedan usar como claves de diccionario o como miembros de conjuntos, y que los *BooleanosDifusos* sean inmutables, pero con la provisión de operadores de asignación aumentados (&= y |=) para asegurarse de que sean cómodos de usar.

6.4.6. Creación de Tipos de Datos desde Cero

Crear el tipo BooleanoDifuso desde cero significa que se debe proporcionar un atributo que contenga el valor difuso y todos los métodos que se requieren.

Booleano Difuso

```
class BooleanoDifuso:
 def __init__(self, valor=0.0):
  self.__valor=valor if 0.0 <= valor <= 1.0 else 0.0
```

El atributo *valor* es privado porque se desea que BooleanoDifuso sea inmutable, por lo que permitir el acceso al atributo sería incorrecto. Además, si se da un valor fuera de rango, se obliga a tomar un valor de 0.0 (falso).

El operador lógico más simple es el *not* lógico, para el cual se ha implementado con la inversión bit a bit (~):

Booleano Difuso-not

```
def __invert__(self):
 return BooleanoDifuso(1.0 - self.__valor)
```

El operador AND lógico bit a bit (&) lo proporciona el método especial *__and__()*, y la versión local (&=) la proporciona *__iand__()*:

Booleano Difuso-And

```
def __and__(self, otro):
 return BooleanoDifuso(min(self.__valor, otro.__valor))

def __iand__(self, otro):
 self.__valor = min(self.__valor, otro.__valor)
 return self
```

Comentario

El método especial *__del__(self)* se llama cuando un objeto se destruye al menos en teoría. En la práctica, nunca se puede llamar a *__del__()*, incluso al finalizar el programa. Además, cuando se escribe *del x*, todo lo que sucede es que la referencia al objeto x se elimina y el contador de referencias al objeto x se reduce en 1. Solo cuando este recuento llega a 0 es probable que *__del__ ()* se llame, pero Python no ofrece ninguna garantía de que alguna vez se llame. En vista de esto, *__del__()* rara vez se vuelve a implementar y no se debe usar para liberar recursos, por lo que no es adecuado para cerrar archivos, desconectar conexiones de red, o desconectar conexiones de base de datos. Python proporciona dos mecanismos separados para garantizar que los recursos se liberen correctamente. Una es usar un bloque try -finally como se ha visto previamente, y la otra es usar un objeto de contexto junto con una declaración *with*.

El operador AND bit a bit devuelve un nuevo *BooleanoDifuso* basado en sus operandos, mientras que la versión de asignación aumentada (in situ) actualiza el valor privado. Estrictamente hablando, este no es un comportamiento inmutable, pero coincide con el comportamiento de algunos otros inmutables de Python, como int, donde, por ejemplo, usar += parece que se está cambiando el operando de la izquierda, pero de hecho se liga nuevamente a un nuevo objeto int que contiene el resultado de la suma. En este caso, no es necesario volver a ligar porque se cambió el *BooleanoDifuso*.

La implementación de *__or__()* que proporciona el o(|) bit a bit para *__ior__()* que proporciona el operador |= no se realizan ya que ambos son iguales a los métodos AND equivalentes excepto que se toma el valor máximo en lugar del valor mínimo de *self* y *other*.

A continuación se muestra una implementación de *__repr__()* que produce una representación compatible con *eval()*.
Por ejemplo, dado *f=BooleanoDifuso.BooleanoDifuso(.75)*; *repr(f)* producirá la cadena *BooleanoDifuso(0.75)*.

Booleano Difuso-repr

```
def __repr__(self):
 return ("{0}({1})".format(self.__class__.__name__,
            self.__valor))
```

Todos los objetos tienen algunos atributos especiales proporcionados automáticamente por Python, uno de los cuales se llama *__class__*, una referencia a la clase del objeto. Todas las clases tienen un atributo *__name__* privado, nuevamente proporcionado automáticamente. Se han usado estos atributos para brindar el nombre de clase para la representación. Esto significa que si la clase *BooleanoDifuso* tiene subclases que agregan métodos adicionales, el método *__repr__()* heredado funcionará correctamente sin necesidad de volver a implementarlo, ya que tomará el nombre de clase de la subclase.

Para la forma de string, simplemente se devuelve el valor de punto flotante formateado como una cadena. No se tiene que usar *super()* para evitar la recursividad infinita porque se llama a *str()* en el atributo *self.__valor*, no en la instancia misma.

Booleano Difuso-str

```
def __str__(self):
 return str(self.__valor)
```

El método especial *__bool__()* convierte la instancia en un valor boo-

leano, por lo que siempre debe devolver *True* o *False*. El método especial *__int__()* proporciona conversión de enteros. Se ha utilizado la función integrada *round()* porque *int()* simplemente se trunca (por lo que devolverá 0 para cualquier valor de *BooleanoDifuso* excepto 1.0). La conversión a punto flotante es sencilla porque el valor ya es un número de punto flotante.

Booleano Difuso-bool-int-float

```
def __bool__(self):
 return self.__valor > 0.5

def __int__(self):
 return round(self.__valor)

def __float__(self):
 return self.__valor
```

Para proporcionar el conjunto completo de comparaciones (<, <=, ==, !=, >=, >) es necesario implementar al menos tres de ellas, <, <= y ==, ya que Python puede inferir >a partir de <, != de == y >= de <=. Se mostrarán solo dos métodos representativos ya que todos ellos son muy similares.

Booleano Difuso- Operadores Relacionales

```
def __lt__(self, otro):
 return self.__valor < otro.__valor

def __eq__(self, otro):
 return self.__valor == otro.__valor
```

Por defecto, las instancias de las clases personalizadas admiten el operador == (que siempre devuelve Falso) y son *hashable* (por lo que pueden ser claves de diccionario y agregarse a conjuntos). Pero si se implementa el método especial *__eq__()* para proporcionar una prueba de igualdad adecuada, las instancias ya no son más *hashable*. Esto se puede arreglar proporcionando un

método especial *__hash__()* como se muestra a continuación.

Booleano Difuso-Hash

```
def __hash__(self):
 return hash(id(self))
```

Python proporciona funciones *hash* para cadenas, números, conjuntos congelados y otras clases. Aquí simplemente se utilizó la función primitiva *hash()* (que puede operar en cualquier tipo que tenga un método especial *__hash__()*) y se le ha dado la ID única del objeto a partir de la cual calcular el *hash*. (No se puede usar el valor privado *self.__valor*, ya que puede cambiar como resultado de una asignación aumentada, mientras que el valor hash de un objeto nunca debe cambiar).

La función primitiva *id()* devuelve un entero único para el objeto que se le da como argumento. Este entero suele ser la dirección del objeto en la memoria, pero todo lo que se puede suponer es que no hay dos objetos que tengan la misma ID.

La función primitiva *format()* es realmente necesaria en las definiciones de clase. Toma un solo objeto y una especificación de formato opcional y devuelve una cadena con el objeto en el formato adecuado.

Cuando se usa un objeto en una cadena de formato, se llama al método *__format__()* del objeto con el objeto y la especificación de formato como argumentos. El método devuelve la instancia con el formato adecuado como describió con anterioridad.

Booleano Difuso-format

```
def __format__(self, format_esp:
 return format(self.__valor, format_esp)
```

Todas las clases integradas ya tienen métodos *__format__()* adecuados; aquí se hace uso del método *float.__format__()* pasando el valor de punto flotante y la cadena de formato que se ha dado. Se podría haber logrado exactamente lo mismo escribiendo:

Booleano Difuso-format

```
def __format__(self, format_esp):
 return self.__valor.__format__(format_esp)
```

El uso de la función *format()* requiere escribir un poco menos y es más claro de leer. Nada obliga a usar la función *format()* en absoluto, así que es posible inventar un lenguaje particular lenguaje de especificación de formato e interpretarlo dentro del método *__format__()*, siempre y cuando se devuelva una cadena.

La función integrada *staticmethod()* es un decorador. Los métodos estáticos son simplemente métodos que no tienen receptor ni argumentos especialmente pasados por Python.

Booleano Difuso-Conjunción

```
@staticmethod
def conjuncion(*difusos):
 return
  BooleanoDifuso(min([float(x) for x in difusos]))
```

El operador & se puede encadenar, por lo que dadas las funciones f, g y h de BooleanoDifuso, se puede obtener la conjunción de todos ellas escribiendo f & g & h. Esto funciona bien para un número pequeño de BooleanosDifusos, pero si se tiene una docena o más empieza a ser bastante ineficiente ya que cada & representa una llamada de función. Con el método dado aquí se puede lograr lo mismo usando una sola llamada de función de *BooleanoDifuso.BooleanoDifuso.conjunción(f, g, h)*. Esto se puede escribir de manera más concisa usando una instancia de BooleanoDifuso, pero dado que los métodos estáticos no tienen receptor, si se llama a uno usando una instancia y se desea procesar esa instancia se debe pasar explícitamente como parámetro, por ejemplo, *f.conjunction(f, g, h)*[22]. La *disjunción()* se define de manera similar

[22]Este uso es confuso dado que los métodos estáticos no tienen receptor con lo cual se desaconseja su utilización.

cambiando *min()* por *max()*.

Comentarios Importantes

Algunos programadores de Python consideran que el uso de métodos estáticos no es de pitónico, y los usan solo si están convirtiendo código de otro lenguaje (como como C++ o Java), o si tienen un método que no usa self. En Python, en lugar de usar métodos estáticos, generalmente es mejor crear una función de módulo, como se verá en la siguiente subsubsección, o un método de clase, como se explicará ver en la última sección. De manera similar, crear una variable dentro de una definición de clase pero fuera cualquier método crea una variable estática (clase). Para las constantes suele ser más conveniente usar módulos globales privados, pero las variables de clase a menudo pueden ser útiles para compartir datos entre todas las instancias de una clase.

6.4.7. Creación de Tipos de Datos desde otro Tipo de Dato

A continuación se muestra la línea de clase de *BooleanoDdifuso* y su método *__new__()*:

Clase Booleano Difuso

```
class BooleanoDifuso(float):
 def __new__(cls, valor=0.0):
  return super().__new__(cls, \
                         valor if 0.0<= valor <=1.0\
                         else 0.0)
```

Cuando se crea una nueva clase, generalmente es mutable y se basa en *object.__new__()* para crear el objeto sin inicializar sin procesar. Pero en el caso de las clases inmutables, se debe realizar la creación y la inicialización en un solo paso, ya que una vez que se ha creado un objeto inmutable, no se puede cambiar.

El método *__new__()* se llama antes de que se haya creado cualquier objeto (dado que la creación de objetos es lo que hace *__new__())*, por lo que no se le puede pasar un objeto propio ya que aún no existe uno. De hecho, *__new__()* es un método de clase; son similares a los métodos normales, excepto que se llaman sobre la clase en lugar de en una instancia y Python proporciona como primer argumento la clase a la que se llaman. El nombre de variable *cls* para *class* es solo una convención, de la misma manera que *self* es el nombre convencional para el objeto.

Entonces, cuando se ejecuta f = BooleanoDifuso(0.7) Python llama a *BooleanoDifuso.__new__(BooleanoDifuso, 0.7)* y crea un nuevo objeto, sea este objeto *difuso*, y luego llama a *difuso.__init__()* y realizan más inicializaciones, finalmente devuelve un referencia de objeto al objeto *difuso* (es esta referencia de objeto con la que se inicializa f). La mayor parte del trabajo de *__new__()* se pasa a la implementación de la clase base, *objeto.__new__()*; todo lo que se hace es asegurar que el valor esté dentro del rango.

Los métodos de clase se configuran utilizando la función integrada *classmethod()* utilizada como un decorador. Pero por conveniencia no se tiene que escribir *@classmethod* antes de *def __new__()* porque Python conoce que este método siempre es un método de clase.

Ahora que se ha visto un método de clase, se puede aclarar los diferentes tipos de métodos que proporciona Python. Los métodos de clase tienen su primer argumento agregado por Python y es la clase del método; los métodos normales[23] tienen su primera argumento agregado por Python y es la instancia en la que se invocó el método; y los métodos estáticos no tienen un primer argumento agregado. Y todos los tipos de métodos pueden recibir argumentos proporcionados por el programador (como su segundo y posteriores argumentos en el caso de los métodos de clase y normales, y como su primera y subsecuentes argumentos para métodos estáticos).

El método *__invert__(self)* se utiliza para proporcionar soporte para el operador NOT bit a bit ( ) es básicamente el mismo que se presentó con anterioridad. Observe que en lugar de acceder a un atributo privado

[23]Métodos de Instancia.

que contiene el valor del *BooleanoDifuso* se usó *self* directamente. Esto es gracias a que *float* se ha heredado lo que significa que un *BooleanoDifuso* se puede usar donde se espera un *float*, siempre que ninguno de los métodos *no implementados* de *BooleanoDifuso* sea usado, por supuesto.

Clase Booleano Difuso - invert

```
def __invert__(self):
 return FuzzyBool(1.0 - float(self))
```

La lógica para estos métodos es la misma que antes (aunque el código es sutilmente diferente), y al igual que el método *__invert__()*, se puede usar *self* y *otro* directamente como si fueran flotantes. Se ha omitido las versiones OR porque se diferencian solo por sus nombres (*__or__()* y *__ior__()*) y en que usan *max()* en lugar de *min()*.

Clase Booleano Difuso - And

```
def __and__(self, otro):
 return FuzzyBool(min(self, otro))

def __iand__(self, otro):
 return FuzzyBool(min(self, otro))
```

Se debe reimplementar el método *__repr__()* a partir de la versión de la clase base *float.__repr__()*. El método de la clase base simplemente devuelve el número como una cadena, pero lo que se necesita es el nombre de la clase para que la representación sea apropiada para *eval()*. Para el segundo argumento de *str.format()* no se puede simplemente pasar self ya que eso resultará en una recursión infinita de llamadas a *__repr__()*, por lo que se invoca a la implementación de la clase base. No se tiene que volver a implementar el método *__str__()* porque la versión de la clase base, *float.__str__()*, es suficiente y se utilizará en ausencia de una de *BooleanoDifuso.__str__()*.

Clase Booleano Difuso - repr

```
def __repr__(self):
 return ("{0}({1})".format(self.__class__.__name__,
            super().__repr__()))
```

Cuando se usa un flotante en un contexto booleano, es falso si su valor es 0.0 y verdadero de lo contrario. Este no es el comportamiento apropiado para *BooleanoDifuso*, por esta razón se tiene que volver a definir este método. De manera similar, usar *int(self)* simplemente truncaría, convirtiendo todo menos 1.0 en 0, así que aquí se usa *round()* para producir 0 para valores hasta 0,5 y 1 para valores hasta e incluyendo el máximo de 1,0.

Clase Booleano Difuso - repr

```
def __bool__(self):
 return self > 0.5

def __int__(self):
 return round(self)
```

No se volverá a implementar el método *__hash__()*, el método *__format__()*, o cualquiera de los métodos que proporcionan los operadores de comparación, ya que todos los provee la clase base *float* funcionan correctamente para *BooleanosDifusos*.

Los métodos que se reimplementaron proporcionan una implementación completa de la clase *BooleanoDifuso*, y han requerido mucho menos código que la implementación presentada con anterioridad. Sin embargo, esta nueva clase *BooleanoDifuso* ha heredado más de 30 métodos que no tienen sentido para esta clase. Por ejemplo, ninguno de los operadores numéricos y de desplazamiento bit a bit básicos (+, -, *, /, «, », etc.) se puede aplicar con sensatez un *BooleanoDifuso*. Así es como se podría dejar sin implementación a la adición:

Clase Booleano Difuso - Sin Implementación de add

```
def __add__(self, other):
 raise NotImplementedError()
```

También se tendría que escribir el mismo código para los métodos *__iadd__()* y *__radd__()* para prevenir completamente la adición. (Tenga en cuenta que *NotImplementedError* es un excepción estándar y es diferente del objeto *NotImplemented*).

Una alternativa a generar una excepción *NotImplementedError*, especialmente si se desea imitar el comportamiento de las clases integradas de Python, es disparar un error de tipo (*TypeError*). Así es como se puede hacer que *BooleanoDifuso.__add__()* se comporte como clases integradas que se enfrentan a una operación no válida:

Clase Booleano Difuso - Sin Implementación de add

```
def __add__(self, otro):
 raise TypeError("tipos de operandos no soporatados \
                  para +: "
      "'{0}' and '{1}'".format(
    self.__class__.__name__, otro.__class__.__name__))
```

Para las operaciones unarias, se desea eliminar la implementación de una manera que imite el comportamiento de los tipos incorporados, el código es un poco más fácil:

Clase Booleano Difuso - Sin Implementación de add

```
def __neg__(self):
 raise TypeError("Tipo de operando inválido para
 el - unario: '{0}'".format(self.__class__.__name__))
```

Para los operadores de comparación, existe una forma simple. Por ejem-

plo, para desimplementar ==, se escribe:

Clase Booleano Difuso - Sin Implementación de add

```
def __eq__(self, other):
 return NotImplemented
```

Si un método que implementa un operador de comparación (<, <=, ==, !=, >=, >), devuelve el objeto integrado *NotImplemented* y se intenta usar el método, Python primero intentará la comparación inversa haciendo el intercambio de los operandos (en caso de que el otro objeto tenga un método de comparación adecuado ya que el objeto propio no lo tiene), y si eso no funciona, Python genera una excepción *TypeError* con un mensaje que explica que la operación no es compatible con los operandos de los tipos usado. Pero para todos los métodos que no son de comparación que no se desean, se debe generar una excepción *NotImplementedError* o *TypeError* como se hizo con los métodos *__add__()* y *__neg__()* mostrados anteriormente.

Sería tedioso desimplementar cada método que no se desea como se ha hecho con anterioridad, aunque funciona y tiene la virtud de ser fácil de entender. Aquí se verá una técnica más avanzada para no implementar métodos. Aquí está el código para no implementar las dos operaciones unarias que no se desean:

Clase Booleano Difuso - Técnica para Inhibir Implementaciones

```
for nombre, operador in (("__neg__", "-"),
  ("__index__", "index()")):
  mensaje = ("tipo de operando incorrecto
para unario {0}: '{{self}}'"
.format(operador))
exec("def {0}(self): raise TypeError(\"{1}\".format("
"self=self.__class__.__name__))"\
  .format(nombre, mensaje))
```


La función primitiva *exec()* ejecuta dinámicamente el código que se le pasa como parámetro. En este caso se le ha dado una cadena, pero también es posible pasar otros tipos de objetos. De forma predeterminada, el código se ejecuta en el contexto del ámbito que lo encierra, en este caso dentro de la definición de la clase BooleanoDifuso, por lo que las sentencias *def* que se ejecutan crean métodos BooleanoDifuso, que es lo que se desea. El código se ejecuta una sola vez, cuando se importa el módulo correspondiente. Aquí está el código que se genera para la primera tupla (*__neg__*, ¨ - ¨):

Código Generado

```
def __neg__(self):
 raise TypeError("tipo de operando incorrecto para u
nario -: '{self}'".format(
                    self=self.__class__.__name__))
```

Se ha hecho que la excepción y el mensaje de error coincidan con los que usa Python para sus propios tipos. El código para manejar métodos binarios y funciones n-arias (como *pow()*) sigue un patrón similar pero con un mensaje de error diferente.

Crear clases de la forma en que se hizo para la primera implementación de *BooleanoDifuso* es mucho más común y es suficiente para casi todos los propósitos. Sin embargo, si se necesita crear una clase inmutable, la forma de hacerlo es volver a implementar *object.__new__()* habiendo heredado uno de los tipos inmutables de Python, como *float*, *int*, *str* o *tuple*, y luego implementar todos los demás métodos que se necesitan. La desventaja de hacer esto es que es posible que se requiera anular algunos métodos; esto rompe el polimorfismo, por lo que en la mayoría de los casos usar la agregación como se hizo en la primera implementación de *BooleanoDifuso* es un enfoque mucho mejor.

6.5. Creación de Colecciones Personalizadas

En las subsecciones de esta sección, se estudiarán las clases personalizadas que son responsables de administrar grandes cantidades de datos. La primera clase que se analizará, *Imagen*, es una que contiene datos de imagen. Esta clase es típica de muchas clases personalizadas que contienen datos en el sentido de que no solo proporciona acceso en memoria a sus datos, sino que también tiene métodos para guardar y cargar los datos hacia y desde el disco. Las clases segunda y tercera que se estudiarán, *SortedList* y *SortedDict*, llenan un vacío raro y sorprendente en la biblioteca estándar de Python para tipos de datos de colección ordenados intrínsecamente.

Creación de Clases que Agregan Colecciones

Una forma sencilla de representar una imagen en color 2D es como una matriz bidimensional en la que cada elemento de la matriz es un color. Entonces, para representar una imagen de 100 × 100, se debe almacenar 10 000 colores.

Para la clase *Imagen*, se tomará una aproximación potencialmente más eficiente. Una imagen almacena un único color de fondo, además de los colores de los puntos de la imagen que difieren del color de fondo. Esto se hace mediante el uso de un diccionario como una especie de matriz rala, en la que cada clave es una coordenada (x, y) y el valor correspondiente es el color de ese punto. Si se tiene una imagen de 100 × 100 y la mitad de sus puntos fueran el color de fondo, se necesitan almacenar solo 5 000 + 1 colores, un ahorro considerable en la memoria.

En el cuadro Errores sólo se muestran las dos primeras clases de excepción; las demás (*ErrorAlCargar*, *ErrorAlGrabar*, *ErrorAlExportar* y *ErrorNombreDeArchivo*) se crean de la misma manera y heredan de *ErrorDeImagen*. Los usuarios de la clase *Imagen* pueden optar por probar cualquiera de las excepciones específicas, o solo la excepción *ErrorDeImagen* de la clase base.

Errores

```
class ErrorDeImagen(Exception): pass
class ErrorDeCoordenada(ErrorDeImagen): pass
```

A continuación se muestran como se intenta usar la clase *Imagen*.

Uso de Imagen

```
colorDeBorde = "#FF0000"
# Rojo
colorCuadrado = "#0000FF"
# Azul
ancho, alto = 240, 60
medioX, medioY = ancho // 2, alto // 2
imagen = Imagen.Imagen(ancho, alto,"OjoCuadrado.img")
for x in range(ancho):
 for y in range(alto):
  if x<5 or x>=ancho - 5 or y<5 or y >= altro-5:
   imagen[x, y] = colorDeBorde
  elif medioX - 20 < x < medioX + 20 and medioY - 20
       < y < medioY + 20:
   imagen[x, y] = colorCuadrado
imagen.grabar()
imagen.exportar("OjoCuadrado.xpm")
```

Tenga en cuenta que se puede usar el operador de acceso al elemento ([]) para configurar los colores en la imagen. Los corchetes también se emplean para obtener o eliminar (configurar efectivamente el color de fondo) el color en una coordenada particular (x, y). Las coordenadas se pasan en una tupla (gracias al operador coma), igual que si se escribe image[(x, y)]. Lograr este tipo de integración de sintaxis perfecta es fácil en Python: solo se tiene que implementar los métodos especiales apropiados, que en el caso del operador de

acceso a elementos son: *__getitem__()*, *__setitem__()* y *__delitem__()*.

La clase *Imagen* usa cadenas hexadecimales de estilo HTML para representar colores. El color de fondo debe establecerse cuando se crea la imagen; de lo contrario, el valor predeterminado es blanco. La clase *Imagen* guarda y carga imágenes en su propio formato personalizado, pero también puede exportar en el formato .xpm, que es aceptado por muchas aplicaciones de procesamiento de imágenes. A continución se comenzará con la implementación de la clase *Imagen*.

Clase Imagen

```
class Imagen:
 def __init__(self, ancho, alto, nombreDeArchivo="",
fondo="#FFFFFF"):
   self.nombreDeArchivo = nombreDeArchivo
   self.__fondo = fondo
   self.__dato = {}
   self.__ancho = ancho
   self.__alto = alto
   self.__colores = {self.__fondo}
```

Cuando se crea una imagen, el usuario (es decir, el usuario de la clase) debe proporcionar un ancho y un alto, pero el nombre del archivo y el color de fondo son opcionales ya que se proporcionan valores predeterminados. Las claves del diccionario *self.__dato* son coordenadas (x, y) y sus valores son cadenas de colores. El conjunto *self.__colores* se inicializa con el color de fondo; se utiliza para realizar un seguimiento de los colores únicos utilizados por la imagen. Todos los atributos de los datos son privados excepto el nombre del archivo, por lo que se debe proporcionar un medio por el cual los usuarios de la clase puedan acceder a ellos. Esto se hace fácilmente usando propiedades.

Acceso a Atributos

```
@property
def background(self):
return self.__background

@property
def width(self):
 return self.__width

@property
def height(self):
 return self.__height

@property
def colors(self):
 return set(self.__colors)
```

Al devolver un atributo de datos de un objeto, se debe saber si el atributo es de tipo inmutable o mutable. Siempre es seguro devolver atributos inmutables ya que no se pueden cambiar, pero para los atributos mutables es necesario considerar algunas compensaciones. Devolver una referencia a un atributo mutable es muy rápido y eficiente porque no se realiza ninguna copia, pero también significa que el llamador tiene acceso al estado interno del objeto y podría cambiarlo de una manera que invalide el objeto[24]. Una política a considerar es devolver siempre un copia de atributos de datos mutables, a menos que la creación muestre un efecto negativo significativo en el rendimiento. (En este caso, una alternativa a mantener el conjunto de colores únicos sería devolver *set(self.__dato.values()) | {self.__fondo}* cada vez que se necesite el conjunto de colores.

Este método devuelve el color de una determinada coordenada utilizando el operador de acceso al elemento ([]).

[24]También rompe la encapsulación.

Método Especial- __getItem__

```
def __getitem__(self, coordenada):
assert len(coordenada) == 2,
"La coordenada debería ser una dupla"
if (not (0 <= coordenada[0] < self.ancho) or
not (0 <= coordenada[1] < self.alto)):
raise ErrorDeCoordenada(str(coordenada))
return self.__dato.get(tuple(coordenada),
                       self.__fondo)
```

Los métodos especiales para los operadores de acceso a elementos y algunos otros métodos especiales relevantes para la colección se enumeran en la Tabla 6.1.

Se ha optado por aplicar dos políticas para el acceso a los elementos. La primer política es que una condición previa para usar un método de acceso a elementos es que la coordenada que se pasa sea una secuencia de longitud 2 (generalmente una dupla), y se usa una aserción para garantizar este requisito. La segunda política es que se acepta cualquier valor de coordenadas, pero si alguno está fuera de rango, se genera una excepción personalizada.

Se ha utilizado el método *dict.get()* con un valor predeterminado del color de fondo para recuperar el color de la coordenada dada. Esto garantiza que si nunca se ha establecido el color para la coordenada, el color de fondo se devolverá correctamente en lugar de generar una excepción *KeyError*.

En el método *__setitem__()*, si el usuario inicializa el valor de una coordenada en el color de fondo, simplemente se elimina el elemento del diccionario correspondiente, ya que se supone que cualquier coordenada que no esté en el diccionario tiene el color de fondo. Se usa dict.pop() y un segundo argumento ficticio en lugar de usar *del* porque hacerlo evita que se genere un *KeyError* si la clave (coordenada) no está en el diccionario. Si el color es diferente del color de fondo, se configura para la coordenada dada y se lo agrega al conjunto de colores únicos utilizados por la imagen.

Método Especial	Uso	Descripción
__contains__(self, x)	x in y	Retorna True si x está en la secuencia y o x es una clave en el mapeo y.
__delitem__(self, k)	del y[k]	Elimina el elemento k-ésimo de la secuencia y o el elemento con clave k en el mapeo y.
__getitem__(self, k)	y[k]	Devuelve el elemento k-ésimo de la secuencia y o el valor de la clave k en el mapeo y.
__iter__(self)	for x in y: pass	Retorna un iterador para los ítems de la secuencia y o para las claves del mapeo y.
__len__(self)	len(y)	Devuelve el número de elementos en y.
__reversed__(self)	reversed(y)	Devuelve un iterador hacia atrás para los elementos de la secuencia y o las claves del mapeo y.
__setitem__(self, k, v)	y[k] = v	Inicializa el k-ésimo elemento de la secuencia y o el valor de la clave k en el mapeo y, a v.

Cuadro 6.1: Métodos Especiales

Método Especial- __setItem__

```
def __setitem__(self, coordenada, color):
 assert len(coordenada) == 2,
        "La coordenada debería ser una dupla"
 if (not (0 <= coordenada[0] < self.ancho) or
    not (0 <= coordenada[1] < self.alto)):
  raise ErrorDeCoordenada(str(coordenada))
 if color == self.__fondo:
  self.__dato.pop(tuple(coordenada), None)
 else:
  self.__dato[tuple(coordenada)] = color
  self.__colores.add(color)
```

Si se elimina el color de una coordenada, el efecto es hacer que el color de esa coordenada sea el color de fondo. Nuevamente, se usa *dict.pop()* para eliminar el elemento, ya que funcionará correctamente ya sea que un elemento con la coordenada dada esté o no en el diccionario.

Tanto *__setitem__()* como *__delitem__()* tienen el potencial de hacer que el conjunto de colores contenga más colores de los que realmente usa la imagen. Por ejemplo, si se elimina un color único que no es de fondo en un píxel determinado, el color permanece en el conjunto de colores aunque ya no se use. De manera similar, si un píxel tiene un color único que no es de fondo y se establece en el color de fondo, el color único ya no se usa, pero permanece en el conjunto de colores. Esto significa que, en el peor de los casos, el conjunto de colores podría contener más colores de los que realmente usa la imagen (pero nunca menos).

Método Especial- __delItem__

```
def __delitem__(self, coordenada):
 assert len(coordenaa) == 2,
        "La coordenada debería ser una dupla"
 if (not (0 <= coordenada[0] < self.ancho) or
     not (0 <= coordenada[1] < self.alto)):
  raise ErrorDeCoordenada(str(coordenada))
 self.__dato.pop(tuple(coordenada), None)
```

Se ha optado por aceptar la compensación de tener potencialmente más colores en el conjunto de colores de los que realmente se usan con el propósito de obtener un mejor rendimiento, es decir, hacer que configurar y eliminar un color sea lo más rápido posible, especialmente porque se almacenan algunos más dado que los colores no suelen ser un problema. Por supuesto, si se desea asegurar de que el conjunto de colores esté sincronizado, se puede crear un método adicional al que se pueda llamar cuando se desea, o aceptar la sobrecarga y hacer el cálculo automáticamente cuando sea necesario. En cualquier caso, el código es muy simple (y se usa cuando se carga una nueva imagen):

Sincronización de Colores

```
self.__colores = (set(self.__dato.valores()) |
                  {self.__fondo})
```

Esto simplemente sobrescribe el conjunto de colores con el conjunto de colores realmente utilizados en la imagen junto con el color de fondo.

No se elaboró una implementación de *__len__()* ya que no tiene sentido para un objeto bidimensional. Además, no se puede proporcionar una forma de representación ya que una imagen no se puede crear completamente llamando a *Imagen()*, por lo que tampoco se proporcionan implementaciones de *__repr__()* (o *__str__()*). Si un usuario llama a *repr()* o *str()* en un obje-

to Imagen, la implementación de la clase base *object.__repr__()* devolverá una cadena adecuada, por ejemplo, *<Objeto Imagen.Imagen en 0x9c794ac>*. Este es un formato estándar utilizado para objetos no aptos para *eval()*. El número hexadecimal es la identificación del objeto; es único (normalmente es la dirección del objeto en la memoria).

Se desea que los usuarios de la clase *Imagen* puedan guardar y cargar sus datos de imagen, por lo que se proporcionan dos métodos, *save()* y *load()*, para realizar estas tareas. Se ha optado por guardar los datos decapándolos. En Python, el decapado es una forma de serializar (convertir en una secuencia de bytes o en una cadena) un objeto de Python. Lo que es tan poderoso en el decapado es que el objeto decapado puede ser un tipo de datos de colección, como una lista o un diccionario, e incluso si el objeto decapado tiene otros objetos dentro (incluidas otras colecciones, que pueden incluir otras colecciones, etc.), se decapará todo el lote, y sin duplicar objetos que se produzcan más de una vez.

Un *pickle* se puede leer directamente en una variable de Python; no se tiene que hacer ningún análisis u otra interpretación. Por lo tanto, usar *pickles* es ideal para guardar y cargar colecciones de datos ad hoc, especialmente para programas pequeños y para programas creados para uso personal. Sin embargo, los *pickles* no tienen mecanismos de seguridad (ni encriptación, ni firma digital), por lo que cargar un *pickle* que proviene de una fuente no confiable podría ser peligroso. En vista de esto, para los programas que no son puramente para uso personal, es mejor crear un formato de archivo personalizado que sea específico para el programa.

La primera parte de la función *save* se refiere únicamente al nombre del archivo. Si el objeto Imagen se creó sin nombre de archivo y no se ha establecido ningún nombre de archivo, entonces el método save() debe recibir un nombre de archivo explícito (en cuyo caso se comporta como *guardar como* y establece el nombre de archivo utilizado internamente). Si no se especifica ningún nombre de archivo, se utiliza el nombre de archivo actual, y si no hay ningún nombre de archivo actual y no se proporciona ninguno, se genera una excepción.

save

```
def save(self, nombreDeArchivo=None):
 if nombreDeArchivo is not None:
  self.nombreDeArchivo = nombreDeArchivo
 if not self.nombreDeArchivo:
  raise ErrorNoNombreArchivo()
 fh = None
 try:
  dato = [self.ancho, self.alto, self.__fondo,
          self.__dato]
  fh = open(self.nombreDeArchivo, "wb")
  pickle.dump(dato, fh, pickle.HIGHEST_PROTOCOL)
 except(EnvironmentError,pickle.PicklingError)as err:
  raise ErrorDeGrabado(str(err))
finally:
 if fh is not None:
  fh.close()
```

Se crea una lista (dato) que contiene los objetos que se desean guardar, incluyendo el diccionario *self.__dato* de elementos de colores coordinados, pero excluyendo el conjunto de colores únicos, ya que esos datos se pueden reconstruir. Luego se abre el archivo para escribir en modo binario y se llama a la función *pickle.dump()* para escribir el objeto de datos en el archivo.

El módulo *pickle* puede serializar datos usando varios formatos (llamados protocolos en la documentación), los cuales se especifican en el tercer argumento de *pickle.dump()*. El protocolo 0 es ASCII y es útil para la depuración. Se ha utilizado el protocolo 3 (*pickle.HIGHEST_PROTOCOL*), un formato binario compacto razón por la cual se abrió el archivo en modo binario. Al leer *pickles*, no se especifica ningún protocolo: la función *pickle.load()* es lo suficientemente inteligente como para resolver el protocolo por sí misma.

read

```
def load(self, nombreDeArchivo=None):
 if nombreDeArchivo is not None:
  self.nombreDeArchivo = nombreDeArchivo
 if not self.nombreDeArchivo:
  raise ErrorNoNombreDeArchivo()
 fh = None
 try:
  fh = open(self.nombreDeArchivo, "rb")
  dato = pickle.load(fh)
  (self.__ancho, self.__alto, self.__fondo,
   self.__dato) = dato
  self.__colores = (set(self.__dato.valores()) |
                    {self.__fondo})
 except(EnvironmentError,pickle.UnpicklingError)
        as err:
  raise ErrorDeCarga(str(err))
 finally:
  if fh is not None:
   fh.close()
```

Esta función comienza igual que la función *save()* intentando capturar el nombre del archivo a cargar. El archivo debe abrirse en modo binario de lectura, y los datos se leen usando la instrucción única *data = pickle.load(fh)*. El objeto de dato es una reconstrucción exacta del que se guardo, por lo que en este caso es una lista con los enteros de ancho y alto, la cadena de color de fondo y el diccionario de elementos de color de coordenada. Se usa el desempaquetado de tuplas para asignar cada uno de los elementos de la lista de datos a la variable apropiada, de modo que cualquier dato de imagen previamente guardado se pierda (correctamente).

El conjunto de colores únicos se reconstruye haciendo un conjunto de todos los colores en el diccionario de colores coordinados y luego agregando

el color de fondo.

export

```
def export(self, nombreDeArchivo):
 if nombreDeArchivo.lower().endswith(".xpm"):
  self.__export_xpm(nombreDeArchivo)
 else:
  raise ExportError("Formato no soportado: " +
   os.path.splitext(nombreDeArchivo)[1])
```

Se ha proporcionado un método de exportación genérico que usa la extensión de archivo para determinar a qué método privado llamar, o genera una excepción para los formatos de archivo que no se pueden exportar. En este caso, solo admite guardar en archivos .xpm (y solo para imágenes con menos de 8930 colores).

Se ha completado la cobertura de la clase personalizada Imagen. Esta clase es típica dado que contiene datos específicos de un programa, brinda acceso a los elementos de datos, posee capacidad de guardar y cargar todos sus datos hacia y desde el disco, entre otras funcionalidades.

6.5.1. Creación de Clases de Colección Usando Agregación

En esta subsección, se desarrollará un tipo de datos de colección personalizado completo, *SortedList*, que contiene una lista de elementos ordenados. Los elementos se ordenan usando su operador menor que (<), proporcionado por el método especial *__lt__()*, o usando una función *Clave* si se proporciona una. La clase intenta hacer coincidir la API de la clase integrada *Lista* para que sea lo más fácil de aprender y usar, pero algunos métodos no se pueden redefinir dado que no aplican respecto del tipo; por ejemplo, usar el operador de concatenación (+) podría dar como resultado que los elementos queden fuera de orden, por lo que no se lo implementará.

Como siempre, cuando se crean clases personalizadas, se tienen las opcio-

nes de heredar una clase que es similar a la que se desea hacer, o crear una clase desde cero y agregar instancias de cualquier otra clase que se necesite dentro de ella, o hacer una mezcla de ambos. Para *SortedList* de esta subsección, se usa la agregación (y se hereda implícitamente de *object*), y para *SortedDict* de la siguiente subsección, se usará tanto la agregación como la herencia (heredando de *dict*).

Estos son algunos ejemplos básicos del uso de una lista ordenada:

SortedList

```
letras = SortedList.SortedList
    (("H", "c", "B", "G", "e"), str.lower)
# str(letras) == "['B', 'c', 'e', 'G', 'H']"
letras.add("G")
letras.add("f")
letras.add("A")
# str(letras) == "['A', 'B', 'c', 'e', 'f', 'G',
                  'G', 'H']"
letras[2] # retorna: 'c'
```

Un objeto *SortedList* agrega (se compone de) dos atributos privados; una función, *self.__key()* (almacenada en *self.__key*), y una lista, *self.__list*. La función *key* se pasa como el segundo argumento (o usando el argumento de palabra clave *key* si no se proporciona una secuencia inicial). Si no se especifica ninguna función *key*, se utiliza la siguiente función de módulo privado: *_identidad = lambda x: x*. Esta es la función *identidad*: simplemente devuelve su argumento sin cambios, por lo que cuando se usa como función clave de *SortedList*, significa que la clave de clasificación para cada objeto en la lista es el mismo objeto. El tipo *SortedList* no permite que el operador de acceso al elemento ([]) cambie un elemento (por lo que no implementa el método especial *__setitem__()*), ni proporciona el método *append()* o *extend()* ya que estos podrían invalidar el orden. La única forma de agregar elementos es pasar una secuencia cuando se crea *SortedList* o agregarlos más

tarde usando el método *SortedList.add()*. Por otro lado, se puede usar con seguridad el operador de acceso al elemento para obtener o eliminar el elemento en una posición de índice determinada, ya que ninguna operación afecta el orden, por lo que se implementan los métodos especiales *__getitem__()* y *__delitem__()*.

A continuación se desarrollará la clase método por método, comenzando como de costumbre con la clase y el inicializador:

SortedList

```
class SortedList:
 def __init__(self, secuencia=None, clave=None):
  self.__clave =clave or _identity
  assert hasattr(self.__clave, "__call__")
  if secuencia is None:
   self.__lista = []
  elif (isinstance(secuencia, SortedList) and
        sequence.clave == self.__clave):
   self.__lista = sequence.__lista[:]
 else:
  self.__lista = sorted(list(secuencia),
                        clave=self.__clave)
```

Dado que el nombre de una función es una referencia de objeto (a su función), se puede mantener funciones en variables como cualquier otra referencia de objeto. Aquí, la variable privada *self.__clave* contiene una referencia a la función *clave* que se pasó o a la función de identidad. La primera sentencia del método se basa en el hecho de que el operador *or* devuelve su primer operando si es *True* en un contexto booleano (que es una función de *key* que no retorna *None*), o su segundo operando en caso contrario. Una alternativa un poco más larga pero más obvia es *self.__clave = clave if clave is not None else _identity*.

Una vez que se tiene la función *clave*, se usa un *assert* para asegurar que se pueda llamar. La función integrada *hasattr()* devuelve True si el objeto

pasado como primer argumento tiene el atributo cuyo nombre se pasa como segundo argumento. Todos los objetos a los que se puede llamar, por ejemplo, funciones y métodos, tienen un atributo *__call__*.

Para que la creación de *SortedLists* sea lo más similar posible a la creación de listas, se tiene un argumento de secuencia opcional que se corresponde con el único argumento opcional que acepta *list()*. La clase *SortedList* agrega una colección de listas en la variable privada *self.__list* y mantiene los elementos en la lista agregada en orden usando la función clave dada.

La cláusula *elif* usa pruebas de tipo para verificar si la secuencia dada es una lista ordenada y, en ese caso, si tiene la misma función clave que esta lista ordenada. Si se cumplen estas condiciones, simplemente se realiza una copia superficial de la lista de secuencias sin necesidad de ordenarla. Si la mayoría de las funciones clave se crean sobre la marcha utilizando lambda, incluso si pueden tener el mismo código, no se compararán como iguales, por lo que es posible que la ganancia de eficiencia no se realice en la práctica. Una vez que se crea una lista ordenada, su función clave es fija, por lo que se mantiene como una variable privada para evitar que los usuarios la cambien. Pero algunos usuarios pueden querer obtener una referencia a la función clave (como se verá en la siguiente subsección), por lo que se ha hecho accesible proporcionando la propiedad clave de solo lectura.

Función Clave

```
@property
def clave(self):
 return self.__clave
```

Cuando se llama al método *add()*, el valor dado debe insertarse en la lista privada *self.__lista* en la posición correcta para preservar el orden de la lista. El método privado *SortedList.__bisect_left()* devuelve la posición del índice donde debe insertarse el elemento. Si el nuevo valor es mayor que cualquier otro valor en la lista, debe ir al final, por lo que la posición del índice será igual a la longitud de la lista (las posiciones del índice de la lista van de 0 a len (L) - 1), si este es en el caso de que se incorpore el nuevo valor. De lo

contrario, se inserta el nuevo valor en la posición del índice, que estará en la posición de índice 0 si el nuevo valor es más pequeño que cualquier otro valor en la lista.

add

```
def add(self, valor):
 índice = self.__bisect_left(valor)
 if índice == len(self.__lista):
  self.__lista.append(valor)
 else:
  self.__lista.insert(índice, valor)



def __bisect_left(self, valor):
 clave = self.__clave(valor)
 izquierdo, derecho = 0, len(self.__lista)
 while izquierdo < derecho:
  medio = (izquierdo + derecho) // 2
  if self.__clave(self.__lista[medio]) < clave:
   izquierdo = medio + 1
  else:
   derecho = medio
 return izquierdo
```

El método privado *__bisect_left(self,valor)* calcula la posición del índice donde pertenece el valor dado en la lista, es decir, la posición del índice donde está el valor (si está en la lista), o donde debería ir (si no está en la lista). Calcula la clave de comparación para el valor utilizando la función clave de la lista ordenada y compara la clave de comparación con las claves de comparación calculadas de los elementos que examina el método. El algoritmo utilizado es la búsqueda binaria, que tiene un rendimiento muy bueno incluso en listas muy grandes; por ejemplo, como máximo, se requieren 21 compa-

raciones para encontrar la posición de un valor en una lista de 1.000.000 de elementos. Compare esto con una lista sin ordenar que utiliza la búsqueda lineal y necesita un promedio de 500.000 comparaciones y, en el peor de los casos, 1.000.000 de comparaciones, para encontrar un valor en una lista de 1.000.000 de elementos.

El método *remove(self,valor)* se utiliza para eliminar la primera aparición del valor dado. Utiliza el método *SortedList.__bisect_left()* para encontrar la posición del índice al que pertenece el valor y luego comprueba si esa posición del índice está dentro de la lista y si el elemento en esa posición es el mismo que el valor dado. Si se cumplen las condiciones, se elimina el elemento; de lo contrario, se genera una excepción *ValueError* (que es lo que hace *list.remove()* en las mismas circunstancias).

remove

```
def remove(self, valor):
 índice = self.__bisect_left(valor)
 if índice < len(self.__lista) and
    self.__lista[índice] == value:
  del self.__lista[índice]
 else:
  raise ValueError("{0}.remove(x):x no está en lista"
                   .format(self.__class__.__name__))
```

El método *remove_ every(self,valor)* es similar al método *SortedList.remove()* y es una extensión de la API de lista. Comienza buscando la posición del índice a la que pertenece la primera aparición del valor en la lista, y luego realiza un ciclo siempre que la posición del índice esté dentro de la lista y el elemento en la posición del índice sea el mismo que el valor dado. El código es ligeramente sutil ya que en cada iteración se elimina el elemento coincidente y, como consecuencia, después de cada eliminación, el elemento en la posición de índice es el elemento que siguió al elemento eliminado.

remove_every

```
def remove_every(self, valor):
 contar = 0
 índice = self.__bisect_left(valor)
 while (índice < len(self.__lista) and
  self.__lista[índice] == valor):
  del self.__lista[índice]
  contar += 1
 return contar
```

El método *count* devuelve el número de veces que aparece el valor dado en la lista (que podría ser 0). Utiliza un algoritmo muy similar a *SortedList.remove_every()*, solo que aquí se incrementa la posición del índice en cada iteración.

count

```
def count(self, value):
 contar = 0
 indice = self.__bisect_left(valor)
 while (índice < len(self.__lista) and
         self.__lista[índice] == valor):
  índice += 1
  contar += 1
return contar
```

Dado que una lista ordenada está ordenada, se usa una búsqueda binaria para encontrar (o no) el valor en la lista.

index

```
def index(self, valor):
 índice = self.__bisect_left(valor)
 if índice < len(self.__lista) and
    self.__lista[índice] == valor:
  return índice
 raise ValueError("{0}.index(x): x no esta en lista"
                  .format(self.__class__.__name__))
```

El método especial *__delitem__()* proporciona soporte para la sintaxis del L[n], donde L es una lista ordenada y n es una posición de índice de número entero. No se prueba un índice fuera de rango ya que si uno recibe la llamada *self.__lista[índice]* generará una excepción *IndexError*, que es el comportamiento deseado.

__delitem__()

```
def __delitem__(self, índice):
 del self.__lista[índice]
```

El método *__getitem__()* implementa $x = L[n]$, donde L es una lista ordenada y n es un índice.

__getitem__()

```
def __getitem__(self, índice):
 return self.__lista[índice]
```

No se permite que el usuario cambie un elemento en una posición determinada (por lo que L[n] =x no está permitido); de lo contrario, el orden de la lista ordenada podría invalidarse. La excepción *TypeError* es la que se usa para indicar que una operación no es compatible con un tipo de datos en particular.

__setitem__()

```
def __setitem__(self, índice, valor):
raise TypeError("Use add para insertar un valor en
                el lugar correcto")
```

El método *__iter__(self, índice, valor)* es fácil de implementar ya que devuelve un iterador de la lista privada usando la función *iter()*. Este método se utiliza para proporcionar el valor a la sentencia *for*.

__iter__()

```
def __iter__(self):
 return iter(self.__lista)
```

El método *__reversed__()* proporciona soporte para la función *reversed()* para que se pueda escribir, por ejemplo, *for value in reversed(iterable)*.

__reversed__()

```
def __reversed__(self):
 return reversed(self.__lista)
```

El método *__contains__()* proporciona soporte para el operador *in*.

__contains__()

```
def __contains__(self, valor):
 índice = self.__bisect_left(valor)
 return (índice < len(self.__lista) and
         self.__lista[índice] == valor)
```

El método *SortedList.clear()* descarta la lista existente y la reemplaza con una nueva lista vacía. El método *SortedList.pop()* elimina y devuelve el

elemento en la posición indicada por el índice, o genera una excepción *IndexError* si el índice está fuera de rango. Para los métodos *pop()*, *__len__()* y *__str__()*, simplemente se pasa el trabajo al objeto *self.__lista*.

Otros Métodos

```
def clear(self):
 self.__lista = []

def pop(self, índice=-1):
 return self.__list.pop(índice)

def __len__(self):
 return len(self.__lista)

def __str__(self):
 return str(self.__lista)
```

No se implementa el método especial *__repr__()*, por lo que se llamará a la clase base *object.__repr__()* cuando el usuario escriba *repr(L)* y L sea una lista ordenada. Esto producirá una cadena como *<Objeto SortedList.SortedList at 0x97e7cec>*, aunque el ID hexadecimal variará.

No se han implementado los métodos *insert()*, *reverse()* o *sort()* porque ninguno de ellos aplica al tipo que se está definiendo. Si se llama a alguno de ellos, se generará una excepción *AttributeError*.

Si copia una lista ordenada usando el lenguaje L[:], se obtendrá un objeto lista, en lugar de una lista ordenada. La forma más fácil de obtener una copia es importar el módulo de copy y usar la función *copy.copy()*; esta es lo suficientemente inteligente como para copiar una lista ordenada (y las instancias de la mayoría de las demás clases personalizadas) sin ayuda. Sin embargo, se proporciona un método *copy()* explícito:

copy

```
def copy(self):
 return SortedList(self, self.__clave)
```

Al pasar *self* como el primer argumento, se asegura que *self.__lista* simplemente se copie superficialmente en lugar de copiarlo y reordenarlo. (Esto es gracias a la cláusula *elif* de prueba de tipo del método *__init__()*). La ventaja de rendimiento teórico de copiar de esta manera no está disponible para la función *copy.copy()*, pero se puede hacer que esté disponible fácilmente agregando esta línea: *__copy__ = copy*.

Cuando se llama a *copy.copy()*, intenta usar el método especial *__copy__()* del objeto, recurriendo a su propio código si no se proporciona uno. Con esta línea en su lugar, *copy.copy()* ahora usará el método *SortedList.copy()* para listas ordenadas.

6.5.2. Creación de Clases usando Herencia

La clase *SortedDict* que se muestra en esta subsección intenta imitar un diccionario lo más fielmente posible. La principal diferencia es que las claves de *SortedDict* siempre se ordenan en función de una función *clave* específica o de la función de identidad. SortedDict proporciona la misma API que dict (excepto que no tiene un repr() apto para eval()), además de dos métodos adicionales que solo tienen sentido para una colección ordenada. A continuación se muestran algunos ejemplos de uso que dan una idea de cómo funciona *SortedDict*:

```
d = SortedDict.SortedDict(dict(s=1, A=2, y=6), str.lower)
d["z"] = 4
d["T"] = 5
del d["y"]
d["n"] = 3
d["A"] = 17
str(d)
```


```
"{'A': 17, 'n': 3, 's': 1, 'T': 5, 'z': 4}"
```

La implementación de *SortedDict* utiliza tanto la agregación como la herencia. La lista ordenada de claves se agrega como una variable de instancia, mientras que la propia clase *SortedDict* hereda la clase *dict*. A continuación se comienza a explicar la implementación de la clase.

SortedDict

```
class SortedDict(dict):
 def __init__(self, diccionario=None, clave=None,
              **kwargs):
  diccionario = diccionario or {}
  super().__init__(diccionario)
  if kwargs:
   super().update(kwargs)
  self.__claves = SortedList.SortedList(super().
                          claves(), clave)
```

La clase base *dict* se especifica en la línea de clase. El inicializador intenta imitar la función *dict()*, pero agrega un segundo argumento para la función clave. La llamada *super().__init__()* se usa para inicializar *SortedDict* usando el método de clase base *dict.__init__()*. De manera similar, si se han usado argumentos de palabras clave, se emplea el método *dict.update()* de la clase base para agregarlos al diccionario. (Tenga en cuenta que solo se acepta una ocurrencia de cualquier argumento de palabra clave, por lo que ninguna de las claves en los argumentos de palabra clave de kwargs puede ser diccionario o clave).

Se mantiene una copia de todas las claves del diccionario en una lista ordenada almacenada en la variable *self.__claves*. Se pasan las claves del diccionario para inicializar la lista ordenada usando el método *dict.keys()* de la clase base; no se usa *SortedDict.keys()* porque depende de la variable *self.__keys* que existirá solo después de que la *SortedList* de claves se haya creado.

El método *update* se usa para actualizar los elementos de un diccionario con los elementos de otro diccionario, o con argumentos de palabras clave, o ambos. Los elementos que existen solo en el otro diccionario se agregan a este, y para los elementos cuyas claves aparecen en ambos diccionarios, el valor del otro diccionario reemplaza el valor original. Se amplió ligeramente el comportamiento manteniendo la función clave del diccionario original, incluso si el otro diccionario es un *SortedDict*.

update

```
def update(self, diccionario=None, **kwargs):
 if diccionario is None:
  pass
 elif isinstance(diccionario, dict):
  super().update(diccionario)
 else:
  for clave, valor in diccionario.items():
   super().__setitem__(clave, valor)
 if kwargs:
  super().update(kwargs)
 self.__claves = SortedList.SortedList(super().
                    keys(), self.__claves.clave)
```

La actualización se realiza en dos fases. Primero se actualizan los elementos del diccionario. Si el diccionario dado es una subclase de *dict* (que incluye *SortedDict*), se usa el método *dict.update()* de la clase base para realizar la actualización; usar la versión de la clase base es esencial para evitar llamar a *SortedDict.update()* de forma recursiva y pasar en un bucle infinito. Si el diccionario no es un *dict*, se itera sobre los elementos del diccionario y se configura cada par clave-valor individualmente. (Si el objeto del diccionario no es un *dict* y no tiene un método *items()*, se generará una excepción *AttributeError*). Si se han usado argumentos de palabras clave, nuevamente se llama al método *update()* de la clase base para incorporarlos.

Una consecuencia de la actualización es que la lista *self.__claves* queda

obsoleta, por lo que se reemplaza con una nueva *SortedList* con las claves del diccionario (nuevamente obtenidas de la clase base, ya que el método *SortedDict.keys()* se basa en la lista *self. __claves)* la cual se está actualizando), y con la función *clave* de la lista ordenada original.

La API *dict* incluye el método de clase *dict.fromkeys()*. Este método se utiliza para crear un nuevo diccionario basado en un iterable. Cada elemento del iterable se convierte en una clave, y el valor de cada clave es *None* o el valor especificado.

fromkeys

```
@classmethod
def fromkeys(cls, iterable, valor=None, clave=None):
 return cls({c: valor for c in iterable}, clave)
```

Debido a que este es un método de clase, Python proporciona automáticamente el primer argumento y es la clase. Para un *dict* la clase será *dict*, y para un *SortedDict* será *SortedDict*. El valor de retorno es un diccionario de la clase dada. Por ejemplo:

Ejemplo fromkeys

```
class MiDict(SortedDict.SortedDict): pass
d = MiDict.fromkeys("VEINS", 3)
str(d)
# retorna: "{'E': 3, 'I': 3, 'N': 3, 'S': 3, 'V': 3}"
d.__class__.__name__
# retorna: 'MiDict'
```

Entonces, cuando se llama a los métodos de clase heredados, su variable cls se establece en la clase correcta, al igual que cuando se llama a los métodos normales[25] y su variable se establece en el objeto actual. Esto es diferente y mejor que usar un método estático porque un método estático está vinculado

[25]Métodos de Instancia.

a una clase en particular y no sabe si se está ejecutando en el contexto de su clase original o en el de una subclase.

__setItem__()

```
def __setItem__(self, clave, valor):
 if calve not in self:
  self.__claves.add(calve)
 return super().__setitem__(clave, valor)
```

El método *__setItem__()* implementa la sintaxis d[clave] = valor. Si la clave no está en el diccionario, se incorpora a la lista de claves, dejando que *SortedList* la coloque en el lugar correcto. Luego se invoca al método de la clase base y se devuelve su resultado al llamador para permitir el encadenamiento, por ejemplo, *x = d[clave] = valor*.

Tenga en cuenta que en la sentencia *if* se verifica si la clave existe en *SortedDict* usando *not in self*. Dado que *SortedDict* hereda *dict*, se puede usar *SortedDict* donde se espera un *dict* y, en este caso, *self* es *SortedDict*. Cuando se reimplementa los métodos *dict* en *SortedDict*, si se necesita llamar a la implementación de la clase base para que haga parte del trabajo, se debe llamar al método usando *super()*, como se ha hecho en la última declaración de este método; hacerlo de esa manera evita que la reimplementación llame al método a sí mismo y entre en recursión infinita.

No se redefine el método *__getitem__()* ya que la versión de la clase base funciona bien y no tiene efecto en el orden de las claves.

El método *__delitem__()* implementa *del d[clave]*. Si la clave no está presente, la llamada *SortedList.remove()* generará una excepción *ValueError*. Si esto ocurre, se detecta la excepción y se genera una excepción *KeyError* para que coincida con la API de la clase *dict*. De lo contrario, se devuelve el resultado de llamar a la implementación de la clase base para eliminar el elemento con la clave dada del propio diccionario.

__delitem__()

```
def __delitem__(self, clave):
 try:
  self.__claves.remove(clave)
 except ValueError:
  raise KeyError(clave)
 return super().__delitem__(clave)
```

El método *setdefault()* devuelve el valor de la clave dada si la clave está en el diccionario; de lo contrario, crea un nuevo elemento con la clave y el valor proporcionados y devuelve el valor. Para SortedDict, se debe asegurar que la clave se agregue a la lista de claves si la clave aún no está en el diccionario.

__setdefault__()

```
def setdefault(self, clave, valor=None):
 if clave not in self:
  self.__claves.add(clave)
 return super().setdefault(clave, valor)
```

El método *pop* funciona de la siguiente manera. Si la clave dada está en el diccionario, este método devuelve el valor correspondiente y elimina el elemento clave-valor del diccionario. La clave también debe eliminarse de la lista de claves.

La implementación es bastante sutil porque el método *pop()* debe admitir dos comportamientos diferentes de forma tal de coincidir con *dict.pop()*. El primero es *d.pop(k)*; aquí se devuelve el valor de la clave k, o si no hay clave k, se genera un *KeyError*. El segundo es *d.pop(k, valor)*; aquí se devuelve el valor de la clave k, o si no hay clave k, se devuelve el valor (que podría ser Ninguno). En todos los casos, si existe la clave k, se elimina el elemento correspondiente.

pop()

```
def pop(self, clave, *args):
 if clave not in self:
  if len(args) == 0:
   raise KeyError(clave)
  return args[0]
 self.__claves.remove(clave)
 return super().pop(clave, args)
```

El método *dict.popitem()* elimina y devuelve un elemento de clave-valor aleatorio del diccionario. Primero se llama a la versión de la clase base, ya que no se sabe de antemano qué elemento se eliminará. Se elimina la clave de la lista de claves y luego se devuelve el ítem.

popitem()

```
def popitem(self):
 item = super().popitem()
 self.__claves.remove(item[0])
 return item
```

El método *clear* borra todos los elementos del diccionario y todos los elementos de la lista de claves.

popitem()

```
def clear(self):
 super().clear()
 self.__claves.clear()
```

Los diccionarios tienen cuatro métodos que devuelven iteradores: *dict.values()* para los valores del diccionario, *dict.items()* para los elementos clave-valor del diccionario, *dict.keys()* para las claves y el método especial *__iter__()*

que brinda soporte para la sintaxis *iter(d)* y opera en las claves (En realidad, las versiones de clase base de estos métodos devuelven vistas de diccionario, pero para la mayoría de los propósitos, el comportamiento de los iteradores implementados aquí es el mismo).

Dado que el método *__iter__()* y el método *keys()* tienen un comportamiento idéntico, en lugar de implementar *keys()*, simplemente se crea una referencia de objeto llamada keys y se configura para que se refiera al método *__iter__()*. Con esto en su lugar, los usuarios de SortedDict pueden llamar a *d.keys()* o *iter(d)* para obtener un iterador sobre las claves de un diccionario, al igual que pueden llamar a *d.values()* para obtener un iterador sobre los valores del diccionario.

No se proporciona una representación para *eval()* de un *SortedDict* porque no es posible producir una representación para *eval()* de la función clave. Entonces, para la reimplementación de *__repr__()*, se omite *dict.__repr__()* y, en su lugar, se llama a la última versión de la clase base, *object.__repr__()*. Esto produce una cadena del tipo que se usa para las representaciones no aptas para *eval()*, por ejemplo, *<Objeto SortedDict.SortedDict en 0xb71fff5c>*.

__repr__()

```
def __repr__(self):
 return object.__repr__(self)
```

Se ha implementado el método *SortedDict.__str__()* porque se pretende que la salida muestre los elementos en ordenados por la clave.

__str__()

```
def __str__(self):
 return ("{" + ", ".join(["{0!r}: {1!r}".format(k, v)
         for k, v in self.items()]) + "}")
```

El método podría haberse escrito así en su lugar:

__str__()

```
items = []
for clave, valor in self.items():
 items.append("{0!r}: {1!r}".format(clave, valor))
return "{" + ", ".join(items) + "}"
```

Los métodos de la clase base *dict.get()*, *dict.__getitem__()* (para la sintaxis v = d[k]), *dict.__len__()* (para *len(d)*) y *dict.__contains__()* (para *x in d*) todos funcionan bien tal como están y no afectan el orden de las claves, por lo que no se implementan nuevamente.

El último método *dict* que se redefine es *copy()*.

copy()

```
def copy(self):
 d = SortedDict()
 super(SortedDict, d).update(self)
 d.__claves = self.__claves.copy()
 return d
```

La implementación más fácil es *def copy (self): return SortedDict(self)*. Se implementó una solución un poco más complicada que evita volver a ordenar las claves ya ordenadas. Se crea un diccionario ordenado vacío, luego se actualiza con los elementos en el diccionario ordenado original usando la clase base *dict.update()* para evitar la reimplementación de *SortedDict.update()*, y se reemplaza el *self.__claves* SortedList del diccionario con una copia superficial de el original.

Cuando se llama a *super()* sin argumentos, funciona en la clase base y el objeto propio. Pero se puede hacer que funcione en cualquier clase y cualquier objeto pasando explícitamente una clase y un objeto. Usando esta sintaxis, la llamada *super()* funciona en la clase base inmediata de la clase que se le da, por lo que en este caso el código tiene el mismo efecto (y podría escribirse

como) *dict.update(d, self)*.

Los dos métodos que se muestran a continuación representan una extensión de la *API dict*. Dado que, a diferencia de un dict simple, un SortedDict está ordenado, se deduce que se aplica el concepto de posiciones de índice clave. Por ejemplo, el primer elemento del diccionario está en la posición de índice 0 y el último en la posición len(d) - 1. Ambos métodos operan en el elemento del diccionario cuya clave está en la posición de índice en la lista de claves ordenadas. Gracias a la herencia, se pueden buscar valores en *SortedDict* utilizando el operador de acceso a elementos ([]) aplicado directamente a *self*, ya que *self* es un *dict*. Si se proporciona un índice fuera de rango, los métodos generan una excepción *IndexError*.

copy()

```
def value_at(self, índice):
 return self[self.__keys[índice]]

def set_value_at(self, índice, valor):
 self[self.__keys[índice]] = valor
```