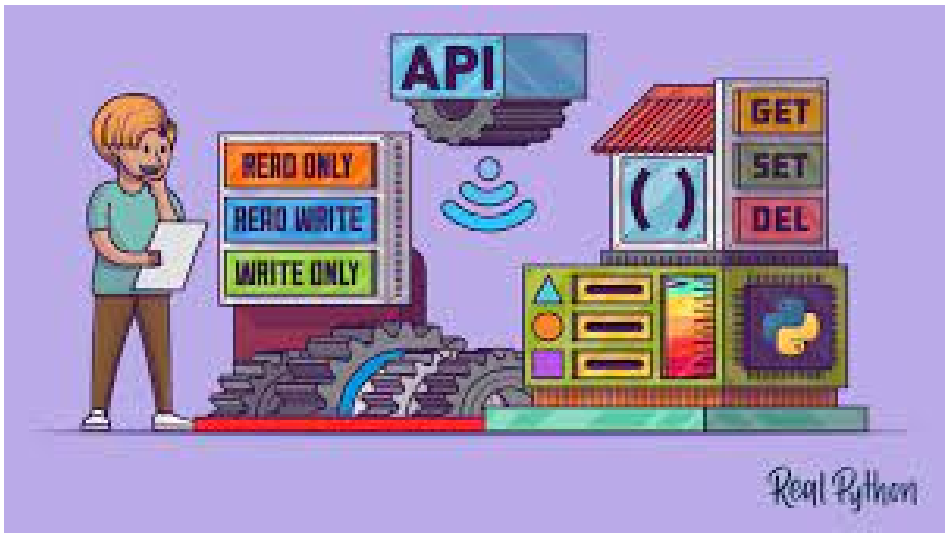
Lenguaje de Programación Python

Soporte Orientado a Objetos: Definición de Tipos Integrados

Dr. Mario Marcelo Berón
Argentina Programa
Universidad Nacional de San Luis

Índice

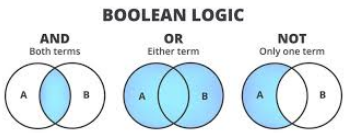
Creación de un Tipo de Dato



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__()*).

Heredar de un tipo de datos existente que sea similar al que se desea crear. Implicar volver a implementar aquellos métodos que se comportan de manera diferente y *ocultar* aquellos métodos que no son requeridos.

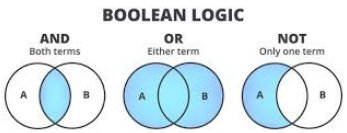
Booleano Difuso - Requisitos



Se desea que el tipo *BooleanoDifuso* admita el conjunto completo de operadores de comparación ($<$, $<=$, $==$, $!=$, $>=$, $>$) y las tres operaciones lógicas básicas, not ($\sim$), and (&), y or (|).

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.

Booleano Difuso - Requisitos



Se quiere proporcionar conversiones a tipos *bool*, *int*, *float*, y *str*, y se desea tener una forma de representación compatible con *eval()*.

BooleanoDifuso admite las especificaciones de formato *str.format()*.

Booleano Difuso - Requisitos

BooleanosDifusos se puedan usar como claves de diccionario o como miembros de conjuntos

Que los *BooleanosDifusos* sean inmutables, pero con la provisión de operadores de asignación aumentados (&= y |=) para asegurarse de que sean cómodos de usar.

Creación de Tipos desde Cero

Importante

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.

Creación de Tipos de Dato desde Cero

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

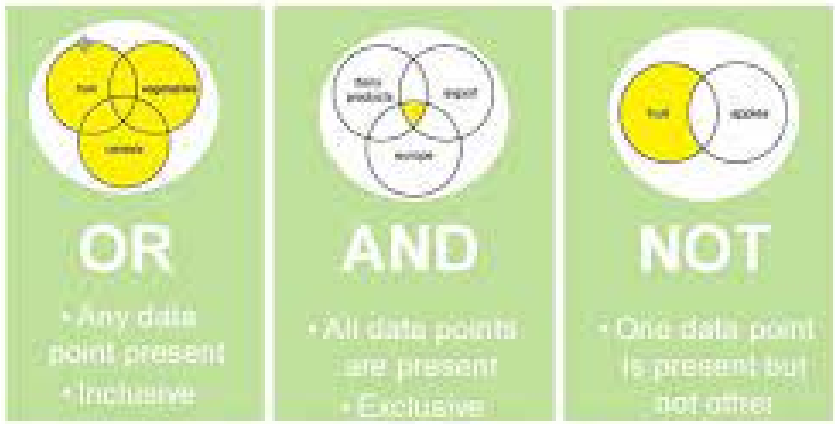
El atributo *valor* es privado porque se desea que BooleanoDifuso sea inmutable, por lo que permitir el acceso al atributo sería incorrecto.

Además, si se da un valor fuera de rango, se obliga a tomar un valor de 0.0 (falso).

Creación de Tipos de Dato desde Cero

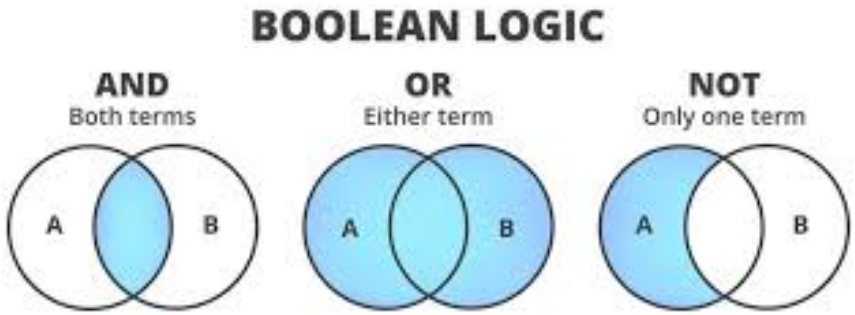
El operador lógico más simple es el *not* lógico, para el cual se ha implementado con la inversión bit a bit ($\sim$).

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



Creación de Tipos de Dato desde Cero

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




```
def __and__(self, other):
 return BooleanoDifuso\
     (min(self.__value, other.__value))
```

```
def __iand__(self, other):
 self.__value = min(self.__value, other.__value)
 return self
```

Creación de Tipos de Dato desde Cero

Comentario

___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. 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.

Comentario

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.

Creación de Tipos de Dato desde Cero

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)*.


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

Creación de Tipos de Dato desde Cero

Para la forma de string, simplemente se devuelve el valor de punto flotante formateado como una cadena.

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

El método especial *__bool__()* convierte la instancia en un valor booleano, por lo que siempre debe devolver *True* o *False*. El método especial *__int__()* proporciona conversión de enteros. La conversión a punto flotante es sencilla porque el valor ya es un número de punto flotante.


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

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

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

Creación de Tipos de Dato desde Cero

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 <=.


```
def __lt__(self, other):
  return self.__value < other.__value

def __eq__(self, other):
  return self.__value == other.__value
```

Creación de Tipos de Dato desde Cero

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__()*.


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

Creación de Tipos de Dato desde Cero

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.

Esto se puede arreglar proporcionando un método especial *__hash__()*.


```
def __format__(self,
               format_spec)
  return format(self.__valor
                format_spec)
```

Creación de Tipos de Dato desde Cero

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


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

Creación de Tipos de Dato a partir de otro Tipo

A continuación la clase *BooleanoDifuso* y el método *__new__()*:

Clase Booleano Difuso

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

__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.

Creación de Tipos de Dato a partir de otro Tipo

El método *__invert__(self)* se utiliza para proporcionar soporte para el operador NOT bit a bit ( ).

Clase Booleano Difuso - invert

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

Observe que en lugar de acceder a un atributo privado que contiene el valor del *BooleanoDifuso* se usó *self* directamente. Esto es gracias a que el *float* heredado lo que significa que un *BooleanoDifuso* se puede usar donde se espera un *float*

Creación de Tipos de Dato a partir de otro Tipo

__and__

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

__iand__

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

Creación de Tipos de Dato a partir de otro Tipo

Comentario Importante

del x

El método especial *___del___(self)* se llama cuando un objeto se destruye al menos en teoría.

del x

En la práctica, nunca se puede llamar a *___del___()*, incluso al finalizar el programa.

del x

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.

Creación de Tipos de Dato a partir de otro Tipo

Comentario Importante

del x

Cuando este recuento llega a 0 es probable que *__del__ ()* se llame.

del x

__del__() rara vez se vuelve a implementar y no se debe usar para liberar recursos.

Creación de Tipos de Dato a partir de otro Tipo

__repr__() que produce una representación compatible con *eval()*.

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.

Creación de Tipos de Dato a partir de otro Tipo

Para la forma de string, simplemente se devuelve el valor de punto flotante formateado como una cadena.

Clase Booleano Difuso - __str__

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

Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-__bool__

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

convierte la instancia en un valor booleano, por lo que siempre debe devolver *True* o *False*.

Booleano Difuso-__int__

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

proporciona conversión de enteros.

Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-__float__

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

La conversión a punto flotante es sencilla porque el valor ya es un número de punto flotante.

Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-__lt__

```
def __lt__(self, other):
return
  self.__value
   < other.__value
```

Implementa $<$.

Booleano Difuso-__eq__

```
def __eq__(self, other):
return
  self.__value
   == other.__value
```

Implementa $==$.

Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-__format__

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

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.

Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-__format__

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

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

Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-__new__

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

En el caso de las clases inmutables, se debe realizar la creación y la inicialización en un solo paso, porque un objeto inmutable, no se puede cambiar.

Creación de Tipos de Dato a partir de otro Tipo

Comentario Importante

Método de Clase

Tienen su primer argumento agregado por Python y es la clase del método

Método de Instancia

Tienen su primer argumento agregado por Python y es la instancia en la que se invocó el método

Método Estático

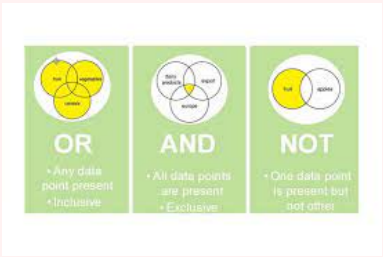
No tienen un primer argumento agregado.

Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-___invert___

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

Se utiliza para proporcionar soporte para el operador NOT bit a bit ($\sim$).

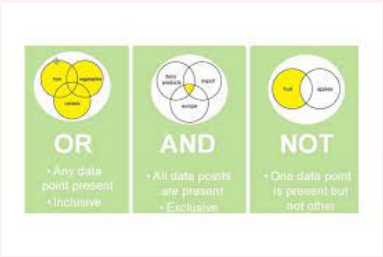


Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-___and___

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

Implementa el operador and.

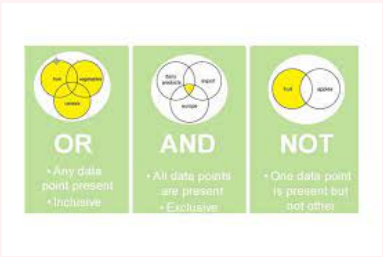


Creación de Tipos de Dato a partir de otro Tipo

__iand__

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

Implementa &=.



Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-__repr__

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

Se debe reimplementar el método *__repr__()* a partir de la versión de la clase base *float.__repr__()*.

Creación de Tipos de Dato a partir de otro Tipo

Comentario Importante

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__()*.

Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-__bool__

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

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 *Booleano-Difuso*, por esta razón se tiene que volver a definir este método.

Creación de Tipos de Dato a partir de otro Tipo

Booleano Difuso-__int__

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

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.

Creación de Tipos de Dato a partir de otro Tipo

Comentario Importante

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*.

Creación de Tipos de Dato a partir de otro Tipo

Comentario Importante

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*.

Creación de Tipos de Dato a partir de otro Tipo

Comentario Importante

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.

Creación de Tipos de Dato a partir de otro Tipo

Inhibir Implementaciones

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