8. Sobrecarga de operadores

May 5, 2014

1 Sobrecarga de operadores

Si tengo mi propio tipo de objeto que representa una estructura matemática (por ejemplo, número complejo, vector) para el cual tiene sentido definir operaciones matemáticas como $+$, $-$, $\times$, $/$, ¿cómo puedo implementar estas operaciones en Python?

Veamos el caso de la diferenciación automática: una manera de derivar una función automáticamente –es decir, de calcular el valor numérico de su derivada en un punto dado.

Vamos a trabajar con *pares* de números $\mathbf{u} = (u, u')$, donde $u : \mathbb{R} \to \mathbb{R}$ y $u = u(a)$, $u' = u'(a)$, donde $a \in \mathbb{R}$. Matemáticamente, hablamos del *jet* de orden 1 (hasta la primera derivada) de la función u en a.

En Python, vamos a definir un nuevo tipo a través una `class`

```
In [11]: class Jet:
             def __init__(self, valor, deriv):
                 self.valor = valor
                 self.deriv = deriv

             def __repr__(self):
                 return "({}, {})".format(self.valor, self.deriv)
```

```
In [12]: j = Jet(3, 4)
```

```
In [13]: j
```

```
Out[13]: (3, 4)
```

Si tengo dos funciones u y v, y quiero calcular la derivada de $u + v$ en a:

$$(u + v)(a) = u(a) + v(a)$$
$$(u + v)'(a) = u'(a) + v'(a)$$

Definir la suma de dos objetos de este tipo:

```
In [14]: def suma(a, b):
             return(a.valor+b.valor, a.deriv+b.deriv)
```

```
In [15]: a = Jet(3, 4)
         b = Jet(4, 5)
```

```
In [16]: suma(a, b)
```

```
Out[16]: (7, 9)
```

Pero esto regresa una tupla, *no* un objeto de tipo `Jet`:

```
In [17]: def suma(a, b):
             return Jet(a.valor+b.valor, a.deriv+b.deriv)

In [20]: suma(a, b)

Out[20]: (7, 9)
```

Pero quiero escribir `a + b` y no `suma(a, b)`: Tenemos que "redefinir" `+` para cuando actúa sobre objetos de tipo `Jet`:

```
In [21]: class Jet:
             def __init__(self, valor, deriv):
                 self.valor = valor
                 self.deriv = deriv

             def __repr__(self):
                 return "({}, {})".format(self.valor, self.deriv)

             def suma(self, b):
                 return Jet(a.valor+b.valor, a.deriv+b.deriv)
```

Ahora tengo que llamar la función así: `a.suma(b)`

```
In [29]: class Jet:
             def __init__(self, valor, deriv):
                 self.valor = valor
                 self.deriv = deriv

             def __repr__(self):
                 return "({}, {})".format(self.valor, self.deriv)

             def __add__(self, otro):
                 return Jet(self.valor+otro.valor,
                            self.deriv+otro.deriv)

             def __mul__(self, otro):
                 return Jet(self.valor*otro.valor,
                            self.valor*otro.deriv + self.deriv*otro.valor)
```

```
In [31]: a = Jet(3, 4)
         b = Jet(4, 5)

         a + b, a*b

Out[31]: ((7, 9), (12, 31))
```

O sea, Python traduce `a+b` a `a.__add__(b)`
Una variable independiente tiene jet:

```
In [33]: a = 3   # punto donde evaluo
         x = Jet(a, 1)

In [34]: x
```

```
Out[34]: (3, 1)

In [35]: x*x

Out[35]: (9, 6)
```

Para manejar constantes, agregamos un valor por defecto ("default") a la constructora:

```
In [41]: class Jet:
             def __init__(self, valor, deriv=0):  # valor por defecto
                 self.valor = valor
                 self.deriv = deriv

             def __repr__(self):
                 return "({}, {})".format(self.valor, self.deriv)

             def __add__(self, otro):
                 return Jet(self.valor+otro.valor,
                            self.deriv+otro.deriv)

             def __mul__(self, otro):

                 if not isinstance(otro, Jet):
                     otro = Jet(otro)

                 return Jet(self.valor*otro.valor,
                            self.valor*otro.deriv + self.deriv*otro.valor)
```

```
In [45]: x = Jet(a, 1)

In [46]: c = Jet(3)

In [47]: c

Out[47]: (3, 0)

In [48]: x * 3

Out[48]: (9, 3)

In [49]: 3 * x
```

```
    ---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

    <ipython-input-49-afb861202d9d> in <module>()
----> 1 3 * x


    TypeError: unsupported operand type(s) for *: 'int' and 'instance'
```

Para hacer la multiplicación al revés (`3*x` en lugar de `x*3`), tengo que definir `__rmul__`:

In []:

```
In [59]: class Jet:
             def __init__(self, valor, deriv=0):  # valor por defecto
                 self.valor = valor
                 self.deriv = deriv

             def __repr__(self):
                 return "({}, {})".format(self.valor, self.deriv)

             def __add__(self, otro):
                 return Jet(self.valor+otro.valor,
                            self.deriv+otro.deriv)

             def __mul__(self, otro):

                 if not isinstance(otro, Jet):
                     otro = Jet(otro)

                 return Jet(self.valor*otro.valor,
                            self.valor*otro.deriv + self.deriv*otro.valor)

             def __rmul__(self, otro):
                 return self * otro



           File "<ipython-input-59-7ee0ddb3ea04>", line 25

         ^
     IndentationError: expected an indented block


In [52]: x = Jet(a, 1)

In [53]: x * 3

Out[53]: (9, 3)

In [54]: 3 * x

Out[54]: (9, 3)
```

Ahora, ¡puedo derivar funciones! Definamos una función normal de Python:

```
In [56]: def f(x):
             return 2*x + 3*x*x

In [58]: a = 3  # derivarla en x=3

         x = Jet(a, 1)  # la derivada de la función "x" es 1

         f(x)

Out[58]: (33, 20)

In []:
```