4_1_AritmeticaDePuntoFlotante

June 9, 2018

[Python Ver.: 3.6.x] | [Autor: Luis Miguel de la Cruz Salas]

1 Aritmética de Punto flotante

- La aritmética que se realiza en una computadora digital es diferente de la que se usa en matemáticas.
  - $2 + 2 = 4$
  - $4^2 = 16$
  - $\left(\sqrt{3}\right)^2 = 3$?

```
In [ ]: 2 + 2
```

```
In [ ]: 4**2
```

```
In [ ]: import math
        math.sqrt(3)**2
```

- En la aritmética continua se permite que un número real pueda tener un número infinito de dígitos.

$$\frac{1}{3} = 0.3333333\ldots 33333\ldots$$

- Una computadora solo puede representar un subconjunto de los números reales, el cual solo contiene números racionales (positivos y negativos).

```
In [ ]: 1/3
```

```
In [ ]: format(1/3, '.52f')
```

- Las computadoras cuentan con una cierta capacidad finita para almacenar información.
- Los números reales se representan mediante los llamados **números de punto flotante** (*floating point numbers*) usando las siguentes características:

  1. Signo **+** o **-** .
  2. Mantisa con t dígitos; donde t es un entero positivo mayor o igual a 1.
  3. Base β; donde β es un entero positivo mayor que 1

4. Exponente e; donde $m \leq e \leq M$ con $m \leq 0$ y $M > t$

- Cada número de punto flotante se representa como:

$$\pm .d_1 d_2 d_3 \ldots d_t \times \beta^e$$

donde $0 \leq d_i \leq \beta - 1 \ \ (i = 1, 2, 3, \ldots, t)$

- La forma normalizada ocurre cuando $d_1 \neq 0$
- El número de dígitos en la mantisa es finito, lo que propicia un error en la representación y en las operaciones aritméticas

1.0.1 Por ejemplo

IBM 3000 series

- Sistema numérico de punto flotante (SNPF) de simple precisión: 1 dígito binario (bit) para el signo, 7 bits para el exponente en base 16, y 24 bits para la mantisa.
  - 24 dígitos binarios corresponde a $\approx$ 6 dígitos decimales.
  - El exponente va de 0000000 = 0 a 1111111 = 127.
  - Para asegurar la representación de números de magnitud pequeña, se resta 64 al exponente, de tal manera que el rango en realidad es de -64 a 63.
  - Por ejemplo:

Signo	e	t
0	1000010	101100110000010000000000

$$1000010 = 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 0^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 66 \Longrightarrow 16^{66-64}.$$

$$\left[\left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^3 + \left(\frac{1}{2}\right)^4 + \left(\frac{1}{2}\right)^7 + \left(\frac{1}{2}\right)^8 + \left(\frac{1}{2}\right)^{14} \right] 16^{66-64} = 179.015625$$

- En este sistema, el siguiente número más pequeño y el siguiente más grande son:

0	1000010	101100110000001111111111	=	179.0156097412109375
0	1000010	101100110000010000000001	=	179.0156402587890625

$$\Longrightarrow 179.015625 \text{ representa } [179.0156097412109375, 179.0156402587890625]$$

- En 1985, el IEEE (Institute for Electrical and Electronic Engineers) publicó: *Binary Floating Point Arithmetic Standard 754, 1985* .

- Se especifican los formatos para precisión simple, doble y extendida, y esos estándares son usados por muchos constructores de CPUs.

```
In [ ]: 1/10 # El resultado que imprime se ve correcto
             # pero el resultado de la operación es
             # la fracción binaria representable más cercana
             # al valor correcto.
```

```
In [ ]: format(1/10, '.52f') # Esto da una idea de lo que está almacenado
```

```
In [ ]: format(0.1, '.52f')
```

```
In [ ]: format(3602879701896397 / 2 ** 55, '.52f')
```

Observamos que el 0.1 y el $\frac{3602879701896397}{2^{55}}$ comparten la misma representación

```
In [ ]: print(str(math.pi))            # Se redondea el valor real almacenado
        print(repr(math.pi))           # para propósitos de despliegue
        print(format(math.pi, '.52f')) #
```

1.0.2 Ejercicio:

£Que resultará de las siguientes evaluaciones?:

```
0.1 == 1/10
0.1 == repr(1/10)
repr(0.1) == 1/10
.1 + .1 + .1
round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
```

Explique el resultado de las evaluaciones.

Hint: Checar el valor más aproximado almacenado en memoria usando por ejemplo `format(0.1,'.53f')`

```
In [ ]: 0.1 == 1/10
```

```
In [ ]: 0.1 == repr(1/10) # Por qué esto da como resultado False?
```

```
In [ ]: repr(0.1) == 1/10 # Por qué esto da como resultado False?
```

```
In [ ]: 0.1 + 0.1 + 0.1 == 0.3
```

```
In [ ]: round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
```

```
In [ ]: round(.1 + .1 + .1, 10) == round(.3, 10)
```

1.1 Algunas funciones útiles

```
In [ ]: x = 3.14159
        x.as_integer_ratio() # regresa los enteros usados para
                             # la representación del número flotante
```

```
In [ ]: 3537115888337719 / 1125899906842624
```

```
In [ ]: x.hex() # Convierte el flotante a hexadecimal
```

```
In [ ]: float.fromhex('0x1.921f9f01b866ep+1') # convierte el hexadecimal a flotante
```

```
In [ ]: x == float.fromhex('0x1.921f9f01b866ep+1')
```

El uso de hexadecimales es útil para portabilidad de valores entre diferentes versiones de Python y para el intercambio de información con otros lenguajes.

- Algunas opciones para realizar las operaciones con *más confianza* en los resultados:
  - La función **round()**
  - El módulo **decimal**
  - El módulo **fractions**

1.2 Round

```
In [ ]: print("(0.1 + 0.1 + 0.1 == 0.3) ?")
        print(round(0.1 + 0.1 + 0.1, 5) == round(0.3, 5))
        print(round(10.33333, 3))
        print(round(1/3,8))
```

```
In [ ]: print('{: .3f}'.format(10.34123)) # £?
```

1.3 Decimal

```
In [ ]: from decimal import getcontext, Decimal
        from math import pi

        print(getcontext())

        print('<'+'-'*20+'>')

        getcontext().prec = 5
        print(Decimal(1)/Decimal(3))

        print('<'+'-'*20+'>')

        getcontext().prec = 30
        print(Decimal(pi))
```

- **getcontext()** permite especificar la precisión y la técnica de redondeo que se usará.
- Por omisión la técnica es ROUND_HALF_EVEN.

2 Fractions

```
In [ ]: from fractions import Fraction

        num1 = Fraction(2,3)
        num2 = Fraction(1,3)

        print("num1 = {} and num2 = {}".format(num1,num2))

        print(num1 + num2)

        print(num1 - num2)

        print(num1*10)

        print(num1/num2)
```

2.1 Características del SNPF en Python 3

NumPy soporta una variedad más amplia de tipos numéricos.

```
In [ ]: import numpy as np
        x = np.float64(0.1)
        print(format(x,'.52f'))
        print(type(x))

        y = np.int_([1,2,4])
        print(y)
        print(type(y))
        print(type(y[1]))

        z = np.arange(3, dtype=np.uint8)
        print(z)
        print(type(z))
        print(type(z[0]))
```

```
In [ ]: xd = math.pi
        print(format(xd,'.52f'))
        print(type(xd))

        xdd = np.float64(xd)
        print(format(xdd,'.52f'))
        print(type(xdd))
```

```
In [ ]: np.finfo(np.float)
```

```
In [ ]: np.finfo(np.float64)
```

```
In [ ]: np.finfo(np.float64).eps
```

```
In [ ]: np.finfo(np.float64).nmant
```

- Para entender esto con mayor detalle véase:
  - http://www.lahey.com/float.htm
  - https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html