Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

27 de agosto de 2021

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro

ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2021
Web: https://github.com/facundobatista/libro-pyciencia

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene dos grandes partes. La primera habla de Python, algunas bibliotecas importantes y otros temas que son fundamentales. En la segunda, mostramos cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En ambos casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

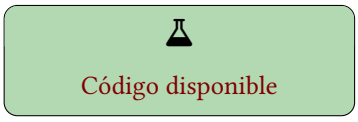
Índice general

Parte I
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos y técnicas fundamentales que serán utilizados en la Parte II al abordar temas de aplicaciones específicas. Se sugiere, a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

1 | Aritmética de punto flotante

Los números de punto flotante se basan en la necesidad de acotar la cantidad de bits necesarios para almacenar el número deseado.



La problemática surge cuando queremos trabajar tanto con números muy grandes como con números muy chicos. Usemos el ejemplo de tener que representar tanto la masa de la Tierra como la del electrón (en gramos, claro). Hacerlo de la siguiente manera representaría un desperdicio de espacio realmente inviable:

```
T = 597200000000000000000000.000000000000000000000000000000000 g
e = 000000000000000000000000.000000000000000000000000000091093 g
```

La solución es utilizar una estructura donde tenemos una "mantisa" multiplicada por la base (10 en el caso de números decimales, 2 en el caso de binarios) elevada a un determinado "exponente":

$$m \times b^n$$

De esta manera ese exponente "nos mueve la coma", y una vez especificada la cantidad de dígitos en la mantisa (que llamamos "precisión") y cómo guardamos el exponente, nos queda una estructura fija que permite guardar esos números ocupando poca cantidad de bits.

Para nuestro ejemplo, supongamos que nuestra estructura de punto flotante tiene una precisión de 5 y la base es 10, por lo que los dos números indicados arriba se convierten en:

$$5.9720 \times 10^{24}$$
$$9.1093 \times 10^{-28}$$

En realidad, la estructura (que incluye el punto decimal) y la base se sobreentienden (están definidas y fijas en el punto flotante utilizado), entonces lo único que guardamos para ambos números son los dígitos `59720` y el exponente `24` en un caso, y los dígitos `91093` y el exponente `-28` en el otro (tener en cuenta que no son números enteros los que guardamos, sino los dígitos en sí). Además también guardamos el signo (que en el ejemplo no vimos porque ambos números son positivos).

Almacenar los números de esta manera nos da varias ventajas, no sólo la de ocupar poco espacio. Es evidente que con esta estructura podemos representar números de órdenes de magnitud muy diferentes (estamos sólo limitados por el exponente, como veremos abajo), y nos proporciona la misma precisión relativa para todos esos números (como mencionamos arriba, limitados por la longitud de la mantisa).

También nos permite algunos cálculos entre esos números muy grandes y muy pequeños (multiplicarlos, por ejemplo), pero tenemos que tener en cuenta que la suma y la resta posiblemente se vean afectadas por la precisión. Siguiendo el ejemplo de arriba, si sumamos la masa de la Tierra y la del electrón, realmente no veremos cambio en el resultado:

$$5972000000000000000000000.000000000000000000000000000091093 = 5.9720 \times 10^{24}$$

Esto es lo que se llama "error de representación" en los números de punto flotante, algo que no podemos evitar mientras manejemos una precisión finita. Tengamos en cuenta que este error no sólo se presenta en casos poco reales como el del ejemplo (nadie anda por la vida sumando la masa de la Tierra y la de un electrón), sino todo el tiempo, en operaciones mucho más triviales. Por ejemplo, en nuestra estructura ficticia "un tercio" sería `33333×10^-5`, y claramente para que el número sea exacto necesitaríamos "infinitos tres" en esa mantisa. Hablaremos más sobre estos errores luego.

También tenemos errores de cálculo cuando llegamos al límite del exponente. Por ejemplo si en nuestra estructura ficticia determinamos que tenemos (además de los cinco para la mantisa) dos dígitos para el exponente, el número más grande que soportará la estructura es el `99999×10^99` y el número más chico el `00001×10^-99`. Si el número es mayor al máximo seguramente tengamos un error de *overflow*, y si el número es menor al mínimo lo más probable es que redondee a cero (aunque los sistemas de manejo de punto flotante permiten mostrar que sucedió un *underflow* e incluso se pueden configurar para que eso sea un error, no un redondeo).

Además, en punto flotante, tenemos los valores especiales "infinito" (positivo y negativo, resultado por ejemplo de dividir algún número por cero), y el valor especial `NaN`, que significa que no es un número (por el inglés "not a number") sino el resultado de una operación indefinida (como multiplicar cero por infinito). Tengamos en cuenta que como mencionaba arriba, muchos de estos casos por default generan un error, pero puede configurarse el sistema para manejarlos.

El tema da para mucho más, se nos escapa del alcance del libro. Para profundizar pueden explorar la página de Wikipedia al respecto [2] o la especificación de aritmética para punto flotante decimal [3], que detalla toda la implementación de un punto flotante de forma genérica sin entrar en la complicación de hacerlo súper optimizado en binario pensado para hardware, como lo hace la norma IEEE 754 [4], para punto flotante binario, que veremos a continuación.

1.1. Punto flotante binario

Hasta ahora hablamos genéricamente de estructuras de punto flotante, y en los ejemplos con los que jugamos usamos números decimales.

Aunque la base puede ser cualquiera, en la práctica se manejan dos grupos, los puntos flotantes decimales y los binarios. Obviamente los primeros son con dígitos en base 10 y los segundos son en base 2.

El punto flotante binario es el punto flotante nativo en todos los lenguajes modernos, debido a que su aritmética está incluida en los procesadores de las computadoras desde hace más de medio siglo, y desde la década de 1990 la mayoría de las implementaciones se basan en el estándar IEEE 754. El punto flotante decimal, aunque hubieron algunos procesadores que lo traian nativo, en la mayoría de los lenguajes está implementado en software, y normalmente siguiendo todos el mismo estándar [5].

En Python al punto flotante binario lo tenemos como tipo de dato integrado en el lenguaje, es el `float`, mientras que al punto flotante decimal lo tenemos implementado en el módulo `decimal`, como ya mencionamos en la subsección de Números **??**.

Ojo, que el punto flotante binario que usa Python, aunque se llame *float*, corresponde en precisión al que en otros lenguajes (como C) se denomina *double* (por "doble precisión", que usa 8 bytes por número), mientras que allí el *float* usa precisión simple, ocupando 4 bytes por número. Mientras que el *float* integrado en Python está bien que sea de doble precisión (porque eso minimiza todo lo posible los errores), en NumPy podemos indicar explícitamente cual queremos utilizar:

CELL 01

```
import numpy as np

np.single
```

```
numpy.float32
```

CELL 02

```
np.double
```

```
numpy.float64
```

Más allá de esos detalles, la característica más importante que debemos mencionar aquí es que el punto flotante binario no puede representar fracciones decimales de forma exacta, por lo que si usamos punto flotante binario no podemos garantizar los mismos resultados que si usáramos aritmética decimal.

Muchas veces no vemos esto porque Python redondea el número usando los 17 dígitos más significativos, entonces si escribimos `1/10` vemos `0.1`, aunque realmente ese número no puede representarse en binario, de la misma manera que no podemos representar `1/3` en notación decimal.

CELL 03

```
1 / 10
```

```
0.1
```

El tipo de datos Decimal, cuando lo inicializamos con un float, nos muestra el número exacto que ese float representa:

CELL 04

```
from decimal import Decimal

Decimal(0.1)
```

```
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

Incluso nos puede sorprender que punto flotante binario represente de la misma manera dos números distintos en decimal:

CELL 05

```
Decimal(0.1)
```

```
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

CELL 06

```
Decimal(0.10000000000000001)
```

```
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

Veamos por ejemplo qué sucede al dividir varias veces un número por diez, primero directa-

mente:

CELL 07

```
x = 9
for _ in range(10):
    print(x)
    x /= 10
```

```
9
0.9
0.09
0.009
0.0009
8.999999999999999e-05
8.999999999999999e-06
8.999999999999999e-07
8.999999999999999e-08
8.999999999999998e-09
```

Esto parece tener mucho sentido, pero si usamos Decimal para ver cual es el número exacto que está manejando el procesador, vemos que son sólo aproximaciones (a veces por arriba, a veces por abajo):

CELL 08

```
x = 9
for _ in range(10):
    print(Decimal(x))
    x /= 10
```

```
9
0.900000000000000022204460492503130808472633361816406 25
0.089999999999999996669330926124530378729104995727539 0625
0.008999999999999999319988397417091618990525603294372558 59375
0.000899999999999999997536692664112933925935067236423492431640625
0.000089999999999999999921156818016854117558978032320737838745117 1875
0.000008999999999999999853394182236510090433512232266366481781005859375
8.999999999999985339418223651009043351223226636648178100585937 5E-7
8.999999999999985339418223651009043351223226636648178100585937 5E-8
8.999999999999978721973323226787644379953690076945349574089050292968 75E-9
```

Entonces, realizar más o menos operaciones vamos incorporando (de mayor o menor manera) un error en el resultado final. Podemos acotar este error, sin embargo, ya que según la implementación del punto flotante binario tendremos un límite superior del error relativo debido al redondeo; este valor se llama "épsilon" y en Python lo podemos encontrar (junto a otros valores que dependen de la implementación) en la estructura `sys.float_info`:

CELL 09

```
import sys
sys.float_info.epsilon
```

```
2.220446049250313e-16
```

En definitiva esto hace que reglas que aprendimos en la escuela como "sumar un número N veces es lo mismo que multiplicar ese número por N" no se cumplen en punto flotante binario

CELL 10

```
0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3
```

```
2.9999999999999996
```

Cell 11

```
0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 == 0.3 * 10
```

```
False
```

Esto es aún más sorpresivo en operaciones más cortas, donde a veces "parece" estar todo bien pero en otros casos, para la misma operación, encontramos que no:

Cell 12

```
5.4 + 0.3 == 5.7
```

```
True
```

Cell 13

```
5.4 + 0.4 == 5.8
```

```
False
```

Por esto es que siempre recomendamos no comparar por igualdad números de punto flotante, sino que lo ideal es evaluar si la diferencia entre los dos números que queremos comparar está acotada a un error máximo que definamos (técnica llamada "comparación con épsilon"):

Cell 14

```
epsilon = 0.00000000001
a = 5.4 + 0.3
b = 5.7
abs(a - b) < epsilon
```

```
True
```

Cell 15

```
a = 5.4 + 0.4
b = 5.8
abs(a - b) < epsilon
```

```
True
```

En estos ejemplos el épsilon fue elegido en función de los números con los que operábamos, pero en realidad no podemos usar siempre un valor absoluto, ya que si los números que estamos comparando son muy pequeños, ese épsilon nos quedó demasiado grande. Por eso siempre se usa un épsilon relativo, en función de los números a comparar. Ojo que este épsilon no es el mismo número que mencionábamos arriba, con el mismo nombre, pero que hacía referencia al máximo error relativo en función de la implementación.

Tenemos que tener en cuenta que todo esto está en la naturaleza básica del punto flotante binario: no es un bug en Python, ni en el procesador, ni en nuestro código, y por eso encontramos el mismo comportamiento en todos los lenguajes que soportan la aritmética de punto flotante de los procesadores (aunque los distintos lenguajes pueden tener distintas reglas sobre cuando y cómo representar estas diferencias).

Estos detalles (y otros) hacen extremadamente difícil desarrollar y probar valores comerciales y financieros, y es por eso que se usa punto flotante decimal en esos casos.

¿Pero qué pasa con aplicaciones científicas?

Cuando tenemos que realizar miles o millones de operaciones, no tenemos alternativa a utilizar punto flotante binario, porque es el que está implementado en hardware: cualquier otra opción sería demasiado lenta.

Entonces, tenemos que lidiar con los errores inherentes a estos números. No los podemos evitar, pero sí podemos controlar su magnitud, no sólo para obtener un resultado final lo más preciso posible, sino también para poder conocer y acotar el error que tenemos al final de una serie de cálculos.

El análisis que debemos hacer se suma, obviamente, al que ya realizábamos sobre los errores de los números originales de esos cálculos (que si son valores teóricos son exactos, pero cuando manejamos mediciones reales, esos mismos números ya vienen con un determinado error que necesitamos conocer).

Por lo general los errores de redondeo son muy pequeños cuando se trabaja con doble precisión y operaciones aisladas. Sin embargo hay situaciones donde pueden aparecer problemas con operaciones simples o al acumularse en cálculos repetidos. La multiplicación y la división son operaciones seguras, pero la suma y la resta son peligrosas (porque para realizarlas hay que operar sobre la mantisa luego de igualar los exponentes, entonces volvemos a tener el problema mostrado al inicio del capítulo cuando mezclamos números de magnitudes muy diferentes).

Veamos un ejemplo de esto, mostrando como no es lo mismo el orden cuando sumamos o restamos:

Cell 16

```
a = 168163.005
b = 168163.004
c = .0000000000123
a - b
```

```
0.0010000000183936208
```

Cell 17

```
a - c - b  # restar "c" se perdió totalmente!!
```

```
0.0010000000183936208
```

Cell 18

```
a - b - c  # acá sí influyó "c"
```

```
0.0010000000060936209
```

Cuantas más operaciones se realice en un cálculo más atención hay que darle a este problema (especialmente con los métodos iterativos de resolución de algoritmos), por eso es que siempre es recomendable utilizar algoritmos diseñados específicamente para minimizar estos errores (particularmente, en lo posible, siempre que se puede utilicemos las bibliotecas de NumPy y SciPy en vez de escribir los algoritmos nosotros).

Para profundizar en el control de error en operaciones aritméticas usando punto flotante les recomendamos el documento “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, de David Goldberg [6].

Parte II
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte I.

Parte III
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [7].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] URL: https://en.wikipedia.org/wiki/Floating-point_arithmetic.

[3] URL: http://speleotrove.com/decimal/decarith.pdf.

[4] URL: https://en.wikipedia.org/wiki/IEEE_754.

[5] URL: http://speleotrove.com/decimal/.

[6] URL: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

[7] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.