

VERSIÓN PRELIMINAR

# Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

7 de noviembre de 2024

VERSIÓN PRELIMINAR

**Título:** Python en Ámbitos Científicos  
**Autores:** Facundo Batista & Manuel Carlevaro  
**ISBN-13 (versión electrónica):** ???-?-???-???-?  
© Facundo Batista & Manuel Carlevaro  
**Primera Edición (versión preliminar)**  
Escrito con X<sub>3</sub>TEX.

Licencia: [Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional \(CC BY-NC-SA 4.0\)](https://creativecommons.org/licenses/by-nc-sa/4.0/)  
Lugar: Olivos y La Plata, Buenos Aires, Argentina  
Año: 2024  
Web: <http://pyciencia.taniquetil.com.ar/>

10 9 8 7 6 5 4 3 2 1

## 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 científicos, 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 tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de los científicos 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.

Tanto los textos como el código fuente, ejemplos e imágenes son Copyright de Facundo Batista y Manuel Carlevaro y están compartidos bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) [1], salvo que se especifique puntualmente lo contrario.

*Olivos y La Plata, Buenos Aires, Argentina,*

---

Facundo Batista & Manuel Carlevaro

# Índice general

<b>Prefacio</b>	<b>2</b>
<b>Índice general</b>	<b>3</b>
<b>I Python</b>	<b>4</b>
<b>II Herramientas fundamentales</b>	<b>5</b>
1. Aritmética de punto flotante	6
1.1. Punto flotante binario . . . . .	7
<b>III Temas específicos</b>	<b>13</b>
<b>IV Apéndices</b>	<b>14</b>
A. Zen de Python	15
<b>Bibliografía</b>	<b>16</b>

## Parte I Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

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.

## Parte II

### Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.





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 traían 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 directamente:

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
0.900000000000000002220446049250313080847263336181640625
0.0899999999999999966693309261245303787291049957275390625
0.0089999999999999931998839741709161899052560329437255859375
0.000899999999999997536692664112933925935067236423492431640625
0.00008999999999999211568180168541175589780323207378387451171875
0.00000899999999999853394182236510090433512232266366481781005859375
8.999999999999853394182236510090433512232266366481781005859375E-7
8.999999999999853394182236510090433512232266366481781005859375E-8
8.9999999999997872197332322678764437995369007694534957408905029296875E-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 sorprendente 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  $\epsilon$ ”):

```
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  $\epsilon$  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  $\epsilon$  nos quedó demasiado grande. Por eso siempre se usa un  $\epsilon$  relativo, en función de los números a comparar. Ojo que este  $\epsilon$  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
<pre>a = 168163.005 b = 168163.004 c = .00000000000123 a - b</pre> <hr/> <pre>0.0010000000183936208</pre>

CELL 17
<pre>a - c - b # restar "c" se perdió totalmente!!</pre> <hr/> <pre>0.0010000000183936208</pre>

CELL 18
<pre>a - b - c # acá sí influyó "c"</pre> <hr/> <pre>0.0010000000060936209</pre>

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 utilizemos 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 III

### 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 [II](#).

**Parte IV**  
**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, ¡tenemos 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](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](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](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/>.