Exactitud

En python podemos definir muchas variables tanto como integer, float, números complejos, listas y arreglos. Todos pueden estar contenidos en la memoria del computador pero tienen límites.

Por ejemplo el número más grande para python es 10^{308} igualmente para los números negativos -10^{308}

Los números exponenciales siempre serán tratados como float

$2e9 = 2 \times 10^{9}$

$1.602e{-}19 = 1.602 \times 10^{-19}$

¿Que sucede si hay un número más grande que 10e308?

Los números que exceden el tamaño máximo son llamados desbordados (overflowed)

Ejemplo: si x es cercano a 10e308 y tratamos de calcular

Y= 10 * x

¿Que creen que pasaría?

Y = inf inf es de “infinite”

Cuando se ejecute un algoritmo y encuentren ese mensaje, se debe verificar el código, pues puede contener errores y los resultados no serán los esperados

En el caso opuesto, cuando el valor es menor a 10e-308 se reconoce a este valor como subdesbordado (underflows), entonces python lo acerca a cero 0.

¿Que sucede con los números integer?

Python guarda cualquier número entero, solo es condicionado por la memoria del computador.

Exercise 4.1: Write a program to calculate and print the factorial of a number entered by the user. If you wish you can base your program on the user-defined function for factorial given in Section 2.6, but write your program so that it calculates the factorial using *integer* variables, not floating-point ones. Use your program to calculate the factorial of 200.

Now modify your program to use floating-point variables instead and again calculate the factorial of 200. What do you find? Explain.

Error numérico

Los números float, a diferencia de los integer, son representados con cierta precisión. Por ahora se reconoce que python considera 16 cifras significativas.

Esto quiere decir que numeros como “pi” o “sqrt(2)” que tienen infinita cantidad de dígitos después de la coma, solo pueden ser representados con aproximación.

True value of π:	3.14159265358979323846264...
Value in Python:	3.141592653589793
Difference:	0.00000000000000238626...

Esta diferencia es llamado **error de redondeo**, este es el error del equipo.

Entonces los cálculos aritméticos de números flotantes solo se garantiza precisión hasta los 16 dígitos después de la coma.

Por ejemplo, si se suma 1.1 y 2.2 el resultado es obviamente 3.3, pero en python realmente es 3.299999999999999.

Probablemente esto no sea crítico en la mayoría de los casos, pero se debe tener en cuenta cuando nuestro algoritmo lo requiera.

Una importante consecuencia del error de redondeo es que **nunca se debe usar *if* para probar la igualdad de dos flotantes.**

```
if x==3.3:
        print(x)
```

Nunca logrará lo que uno lógicamente busca, pues 3.3 no existe como tal y python lo asigna como 3.299999999999999, por lo tanto el programa fallará.

```
Epsilon = 1e-12
If abs(x-3.3)<epsilon:
        print(x)
```

Entonces aquí si puede funcionar la condición, para un epsilon pequeño, donde el número puede ser cercano a 3.3, pero no necesariamente el mismo.

Por lo tanto el valor de epsilon debe ser elegido apropiadamente dependiendo del caso que estemos usando y la precisión que busquemos.

Ejemplo

Escriba un programa que tome como entrada tres números, a, b y c, e imprima las dos soluciones de una ecuación cuadrática de la forma

$$ax^2 + bx + c = 0$$

Que su programa calcule las soluciones para:

$$0.001x^2 + 1000x + 0.001 = 0$$

Una alternativa para escribir las soluciones de una ecuación cuadrática es multiplicando arriba y abajo por $-b \mp \sqrt{b^2 - 4ac}$
Entonces las soluciones pueden ser escritas como:

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}.$$

Añade más líneas al programa para imprimir estos valores, además de los anteriores y de nuevo utilizar el programa para resolver

$$0.001x^2 + 1000x + 0.001 = 0$$

¿Qué ves? ¿Cómo lo explicas?

Velocidad de un programa

Los computadores pueden hacer muchas operaciones matemáticas en un corto tiempo, pero a medida que aumentan las operaciones, estas requieren más tiempo de cómputo.

Un millón de operaciones matemáticas no es problema para un computador, puede realizar esto en menos de un segundo.

Si operamos con un millón de números o un millón de raíces cuadradas, el tiempo va sumando un poco más.

Un billón de operaciones puede tomar minutos u horas

Un trillón de operaciones puede ser infinito

Se recomienda un billón de operaciones o menos.

Ejemplo: Oscilador armónico cuántico a temperatura finita

EXAMPLE 4.2: QUANTUM HARMONIC OSCILLATOR AT FINITE TEMPERATURE

The quantum simple harmonic oscillator has energy levels $E_n = \hbar\omega(n + \frac{1}{2})$, where $n = 0, 1, 2, \ldots, \infty$. As shown by Boltzmann and Gibbs, the average energy of a simple harmonic oscillator at temperature T is

$$\langle E \rangle = \frac{1}{Z} \sum_{n=0}^{\infty} E_n e^{-\beta E_n}, \tag{4.11}$$

where $\beta = 1/(k_B T)$ with k_B being the Boltzmann constant, and $Z = \sum_{n=0}^{\infty} e^{-\beta E_n}$. Suppose we want to calculate, approximately, the value of $\langle E \rangle$ when $k_B T = 100$. Since the terms in the sums for $\langle E \rangle$ and Z dwindle in size quite quickly as n becomes large, we can get a reasonable approximation by taking just the first 1000 terms in each sum. Working in units where $\hbar = \omega = 1$, here's a program to do the calculation:

```
from math import exp
terms = 1000
beta = 1/100
S = 0.0
Z = 0.0
for n in range(terms):
        E = n + 0.5
        weight = exp(-beta*E)
        S += weight*E
        Z += weight
print(S/Z)
```

1. Constantes son definidas al comienzo. Están identificadas en el programa.

2. Se usó solo un **lopp for** para desarrollar la suma, esto hace más rápido el programa.

3. Aunque la exponencial aparece separadamente en ambas sumas, se calculó una vez para cada paso del loop for y se guardó en la variable weight. Esto ahorra tiempo. Las exponenciales toman tiempo en calcular, entonces aquí se redujo a un calculo.

Corramos el programa, el resultado es:

99.9554313409

Aumentemos el número de términos en la suma a un **millón**, el resultado es:

100.000833332

Aumentemos el número de términos en la suma a un **billón**, el resultado es:

100.000833332

¿Como es la exactitud?

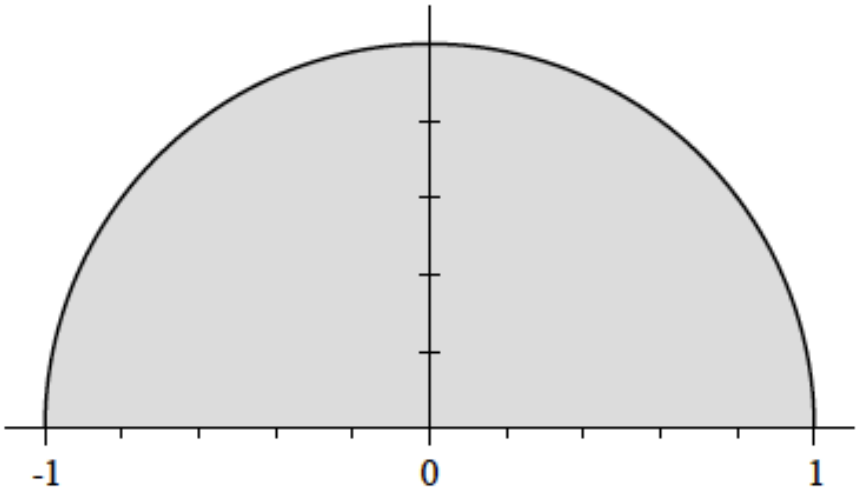
Tip: encontrar equilibrio entre velocidad y exactitud

Exercise 4.4: Calculating integrals

Suppose we want to calculate the value of the integral

$$I = \int_{-1}^{1} \sqrt{1 - x^2}\, dx.$$

The integrand looks like a semicircle of radius 1:



and hence the value of the integral—the area under the curve—must be equal to $\frac{1}{2}\pi = 1.57079632679\ldots$

Alternatively, we can evaluate the integral on the computer by dividing the domain of integration into a large number N of slices of width $h = 2/N$ each and then using the Riemann definition of the integral:

$$I = \lim_{N\to\infty} \sum_{k=1}^{N} h y_k\,,$$

where

$$y_k = \sqrt{1 - x_k^2} \qquad \text{and} \qquad x_k = -1 + hk.$$

We cannot in practice take the limit $N \to \infty$, but we can make a reasonable approximation by just making N large.

a) Write a program to evaluate the integral above with $N = 100$ and compare the result with the exact value. The two will not agree very well, because $N = 100$ is not a sufficiently large number of slices.

b) Increase the value of N to get a more accurate value for the integral. If we require that the program runs in about one second or less, how accurate a value can you get?