ACERCA DEL AUTOR

JOSÉ LUIS PRIETO MORLANÉS

Licenciado en Informática por la Universidad Politécnica de Madrid. Ha pasado por diversos centros de formación (Microsoft, Ericsson y Nokia) como alumno y como profesor.

Trabajó en Ericsson Information Systems (Linköping - Suecia) y Nokia Data (Helsinki), como responsable de soporte de sistemas, participando en uno de los departamentos de I+D de Nokia Data. A lo largo de los años ha usado multitud de lenguajes de programación, Cobol, Ensamblador, Fortran, Algol, Basic, Visual Basic, C, Visual C, C#, JavaScript y Python, siendo este último el que ha más le ha cautivado.

INTRODUCCIÓN

Este libro está centrado en los módulos de Python relacionados con el uso de las matemáticas en la programación, acompañado con ejemplos sencillos y operativos que cubren las distintas funcionalidades que ofrecen para la realización de cálculos numéricos.

HISTORIA

Python es un lenguaje de programación creado por Guido van Rossum a principios de los años 90. La historia del desarrollo de Python se ha mantenido siempre en la línea del Software Libre y de Código Abierto (incluso antes de que existiera este término), por lo que la práctica totalidad de sus numerosas librerías de cálculo científico son libres y gratuitas también. Python ha asimilado otros lenguajes, como el Fortran, que ha sido durante muchos años la referencia de leguaje de programación matemático, integrando sus bibliotecas, adaptándolas a la sintaxis y legibilidad propias de Python.

La primera idea de Python surge en 1982 mientras Guido Van Rossum trabajaba en CWI (Centrum Wiskunde & Informatica, Centro de Matemáticas y Ciencias de la Computación) en Ámsterdam, Holanda, en el equipo de desarrollo del lenguaje ABC. Resulta curioso que el Algol 68 también se desarrolló en el CWI. En 1989 comienza el desarrollo de Python, en los primeros meses de 1990 ya estaba disponible una primera versión operativa. El 20 de febrero de 1991, se lanza la primera versión de Python, la v0.9.0. La versión 2.0 se lanzó en 2000, era ya una versión mucho más madura. En diciembre de 2008, se lanzó la versión 3.0.

El libro está orientado a Python 3.x.

A LOS USUARIOS DEL LIBRO

Este libro no es un curso de Python ni de programación. Se requiere una cierta experiencia en programación y conocimiento del lenguaje Python para hacer uso de los módulos matemáticos aquí expuestos.

Siempre he pensado que la mejor manera de aprender un lenguaje de programación es como aprender a andar, hay que hacerlo con la práctica, así que los ejemplos no están solo para ver como se acumulan las sentencias una tras otra, si no que hay que copiarlos en el sistema de desarrollo y ejecutarlos, y seguirlos paso a paso para ver su funcionamiento. Eso contesta más cómos y porqués que cualquier explicación teórica, por buena que sea.

ORGANIZACIÓN DEL LIBRO

El libro está estructurado en tres partes. La primera, denominada Matemáticas I, incluye los módulos matemáticos generales de Python *math* y *cmath* para números reales y complejos; los módulos *decimal* y *fractions* para el uso de decimales tal y como nos los han enseñado en el colegio y fracciones; siguen los módulos para el uso de números aleatorios *random* y *secrets*; cerrando con el módulo de estadísticas *statistics*.

La segunda parte, bajo el epígrafe de Gráficos, cubre la capacidad de Python de presentar la información numérica mediante gráficos con el módulo *matplotlib*. Somos seres visuales, y el mostrar un gran volumen de datos numéricos de una forma gráfica siempre ayuda a comunicar lo que esos números contienen, favoreciendo además su comprensión.

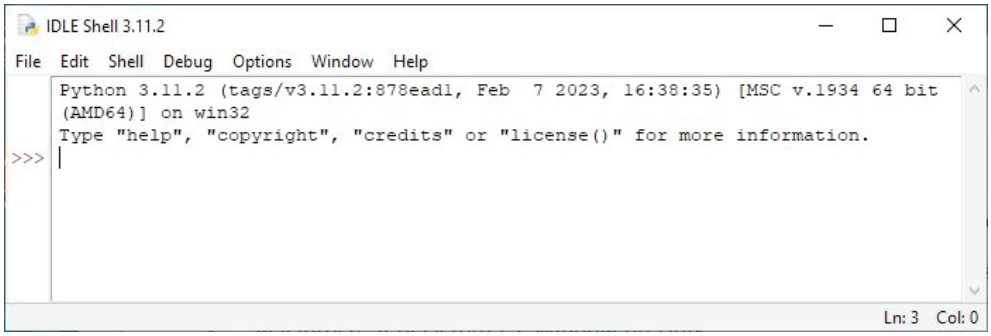
La tercera parte, con el título de Matemáticas II, se centra en uno de los módulos matemáticos avanzados de Python, *numpy*, que facilita el almacenamiento y manipulación de grandes cantidades de información de una manera eficiente

REQUISITOS

Para poder ejecutar los ejemplos del curso se necesita instalar el intérprete de Python 3. Es posible *descargar una copia (https://www.python.org/downloads/)* desde la Organización Python. La versión operativa para Windows en el momento de escribir estas líneas es la Python 3.11. Basta seguir las instrucciones de instalación que van apareciendo.

El paquete de instalación proporciona un entorno de desarrollo denominado IDLE (*Integrated Development and Learning Environment*) Entorno de Aprendizaje y

Desarrollo Integrado, en donde se podrán ejecutar los ejemplos y lanzarse a otras pruebas y desarrollos más avanzados. Una vez arrancado tiene la siguiente apariencia:



IDLE 3.11.2

El IDLE de Python es un editor e intérprete elemental que permite editar y ejecutar programas en Python, además, al ser un entorno interactivo se pueden ejecutar instrucciones sueltas de Python. El editor ofrece un resaltado en color de las palabras clave del lenguaje:

- Las palabras reservadas de Python se muestran en color naranja.
- Las cadenas de texto se muestran en verde.
- Las funciones se muestran en púrpura.
- Los resultados de las órdenes se visualizan en azul.
- Los mensajes de error se muestran en rojo.

Todos los ejemplos del curso han sido probados con el IDLE de Python 3.11.

CONVENCIONES

Se han establecido las siguientes convenciones en la documentación del curso.

La sintaxis del lenguaje se presenta mediante un sombreado en azul, indicando entre corchetes angulares <> los elementos que no pertenecen al lenguaje, de la forma:

```
def <nombre_función>(<parámetros>):
    <sentencias>
```

Donde el literal **def**, los paréntesis y los dos puntos finales son los únicos elementos del lenguaje en sí.

Los ejemplos se presentarán como se pueden ver y ejecutar en el IDLE, con el símbolo indicador (*prompt*) de Python >>>, tal y como aparece en el intérprete de Python. Las

líneas sin el indicador pertenecen a un bloque de código o son la salida de la ejecución del guión, como ocurre en el IDLE.

```
>>> print(“Hola, Mundo”)
Hola, Mundo
>>> if 5>2:
... print(True)
True
```

Todos los ejemplos llevan el resaltado del código en color como se ve en el IDLE de Python.

El resultado de la ejecución se muestra también a continuación del código, de la misma forma que se vería al ejecutarlo.

```
Datos cargados en la matriz
[[  1.    2       3.]
 [ 11. nan    33.]
[ nan 222. 333.]]
```

Recordar que hay muchas formas de codificar un algoritmo, unas claras y otras confusas. La comunidad de Python emplea el término *Pythonic* para referirse al código que sigue un cierto estilo y las convenciones descritas en Guía de estilo *PEP 8 -- Style Guide for Python Code (https://www.python.org/dev/peps/pep-0008/)* con sugerencias sobre cómo escribir un código fuente legible y normalizado. o establecidas por la práctica. Ser *pythonista* se va aprendiendo. El buen código se acaba por reconocer.

Los scripts llevan una marca en la parte superior con el nombre del fichero que se puede descargar desde la web de RA- MA.

matplotlib_00_1_valoresX.py

```
import matplotlib.pyplot as plt


# relación de valores a visualizar
valores = [1, 1, 2, 3, 5, 8, 13, 24]

# dibujar los valores
plt.plot(valores)
# mostrar el gráfico
plt.show()
```

Cuando se lista el contenido de un fichero, este se muestra en una caja con una cabecera con el nombre del fichero.

El contenido del fichero **datos_00.csv** es:

datos_00.csv

```
1, 2, 3
11, 22, 33
111, 222, 333
```

En cualquier caso, lo último de lo último siempre se encontrará en la *documentación oficial de Python (https://docs.python.org/3/).*

1

MATEMÁTICAS

1.1 INTRODUCCIÓN

El cálculo numérico o computación numérica (*numerical computing*) es la rama de las matemáticas encargada de diseñar algoritmos para la solución de problemas en análisis numérico.

El análisis numérico se utiliza para resolver problemas tanto de ciencia como de ingeniería, además de en muchos otros campos. Pocos programas se salvan de contener algún cálculo matemático que complete una tarea en un algoritmo. Desde simples operaciones aritméticas, en las que intervienen los números reales, hasta otras que requieran el uso de números complejos o de funciones trigonométricas.

Aunque Python tiene varios tipos de datos estructurados en la práctica no son nada adecuados para el cálculo numérico. Además, es importante tener en cuenta el rendimiento de los algoritmos tanto en lo que respecta a la velocidad como al uso de los datos.

Python es un lenguaje de propósito general que puede emplearse sin hacer uso de ningún módulo numérico especial, pero que no podría utilizarse para el análisis de datos, la estadística o el aprendizaje automático, sin la aportación de módulos numéricos específicos. En general para la ciencia de datos (*Data Science*), que precisa de la preparación de los datos, su manipulación, clasificación y análisis para extraer la información que contienen y representarla para facilitar su comprensión.

Python, junto con sus módulos *NumPy*, *SciPy*, *Pandas* y *Matplotlib*, se encuentra entre los mejores sistemas de programación numérica. Y a un coste cero.

1.1.1 Aritmética de punto flotante

Los ordenadores sólo pueden almacenar de forma nativa números enteros, por lo que se hace necesaria alguna forma de representación de los números decimales. Esta representación no es perfectamente precisa, así, la mayoría de las veces 0.1 + 0.2 no es 0.3. Podemos verlo sobre el IDLE de Python:

```
>>> print(.1 + .2)
0.30000000000000004
>>> .1 + .1 + .1 == .3
False
>>> print(.1 + .1 + .1)
0.30000000000000004
```

Esto se debe a que la mayoría de las fracciones decimales, en un sistema en base 10, sólo pueden representarse exactamente cuándo utilizan un factor primo de la base.

Los factores primos de 10 son 2 y 5, así que 1/2, 1/4, 1/5, 1/8 y 1/10 pueden expresarse correctamente porque todos los denominadores utilizan factores primos de 10. En cambio, 1/3, 1/6, 1/7 y 1/9 tienen decimales que se repiten, secuencias periódicas, porque sus denominadores utilizan un factor primo de 3 o 7.

Como los números reales (en informática también los denominamos de punto flotante) en la computadora se representan en fracciones en base 2 (binario), y el único factor primo es 2, sólo se pueden expresar exactamente las fracciones cuyo denominador tiene 2 como factor primo, 1/2, 1/4 y 1/8, mientras que 1/5 o 1/10 tienen decimales repetidos.

Como consecuencia de esto 0,1 y 0,2 (1/10 y 1/5), aunque son decimales exactos en un sistema de base 10, no lo son en el sistema de base 2 que utiliza la computadora. E, igualmente, la conversión de números decimales de base 2 a una representación en base 10 nos proporciona aproximaciones, pero no el valor exacto.

Dado que los números reales constituyen un conjunto que no está acotado ni superior ni inferiormente (es infinitamente denso, entre dos números reales siempre hay un número real) y que no es posible representarlo en la memoria del computador que es limitada y no puede almacenar números con una precisión infinita, terminan apareciendo errores de redondeo en los números en punto flotante, y cuando realizamos una secuencia de cálculos con un error de redondeo debido a una representación inexacta los errores se terminan acumulando y magnificando en el resultado final.

De todas formas los errores en las operaciones de punto flotante en casi todas las máquinas están en el orden de 1/2**53. Suficiente para la mayoría de los casos en que empleemos operaciones con decimales.

En el computador, los números reales se representan mediante un formato denominado de punto flotante que utiliza sólo un número finito de dígitos, siguiendo la notación científica, mediante una mantisa M y un exponente E, representados con un punto explícito siempre entre el primer y el segundo dígito.

Este formato fue establecido por el IEEE (*Institute of Electrical and Electronics Engineers* - Instituto de Ingenieros Eléctricos y Electrónicos) en 1985, el estándar **IEEE 754**.

El formato de punto flotante permite representar números de órdenes de magnitud dispares limitado por la longitud del exponente. Proporcionando la misma precisión relativa para todos los órdenes limitado por la longitud de la mantisa.

Para precisión simple emplea 32 bits:

- 1 bit para el signo (0 positivo, 1 negativo)
- 8 bits para el exponente (la característica)
- 23 bits para la parte fraccionaria (la mantisa)

Para precisión doble emplea 64 bits:

- 1 bit para el signo (0 positivo, 1 negativo)
- 11 bits para el exponente (la característica)
- 52 bits para la parte fraccionaria (la mantisa)

Casi todas las plataformas de Python hacen uso de la doble precisión según la norma IEEE754 - 64 bits.

```
1. >>> import sys
2. >>> sys.float_info
3. sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
   min=2.22507
```

En Python, los números que son mayores que el mayor número representable en coma flotante producen un desbordamiento, al que se le asigna el valor de infinito. Los números que son más pequeños que el menor número producen un desbordamiento negativo, al que se asigna el valor de 0.

Python dispone de diferentes módulos para los casos en que precisemos de una representación decimal exacta, como es el módulo *decimal*, para aplicaciones de alta precisión. También el módulo *fractions*, que implementa aritmética basada en números racionales. O el paquete *SciPy*, para operaciones matemáticas y estadísticas con mayores precisiones.

Sirva esto como una breve explicación del porqué existen diferentes módulos para el manejo de números reales, decimales o fracciones en Python.

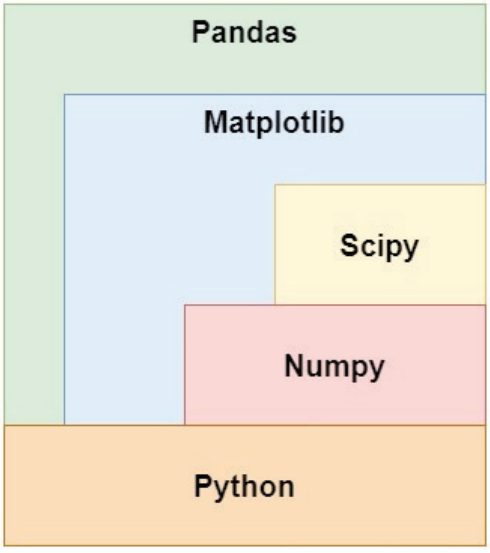
1.1.2 Módulos matemáticos

El uso de Python puro, sin ningún módulo numérico, no podría utilizarse para las tareas numéricas que ofrecen paquetes como Matlab o lenguajes como R. Sin embargo, el uso de los diferentes módulos que ofrece Python para el análisis numérico lo sitúa

al mismo nivel o por delante de otros paquetes. Con la ventaja añadida de que es completamente gratis.

Python dispone de los módulos:

- *Math.* Para las funciones matemáticas comunes con números reales
- *Cmath.* Para trabajar con números complejos
- *Decimal.* Para trabajar con números decimales.
- *Fractions.* Para realizar operaciones con fracciones.
- *Numpy.* Proporciona estructuras de datos básicas, implementando arrays y matrices multidimensionales, con las funciones para crear y manipular estas estructuras de datos.
- *SciPy.* Módulo científico que hace uso de las estructuras de datos proporcionadas por NumPy, completando sus funciones estadísticas.
- *Matplotlib.* Para visualizaciones y gráficos de los módulos numéricos previos.
- *Pandas.* Construido sobre los módulos anteriores ofrece estructuras de datos y operaciones para manipular tablas numéricas y series temporales.
- *Random.* Para la generación de números pseudoaleatorios.
- *Secrets.* Generador de números pseudoaleatorios criptográficamente fuertes.
- *Statistics.* Para realizar cálculos estadísticos sobre datos numéricos.



Relación gráfica entre los módulos matemáticos de Python

2

MÓDULOS MATEMÁTICOS GENERALES

A partir de aquí recorreremos los módulos matemáticos incluidos en la distribución general de Python: *math*, *cmath*, *decimal* y *fractions* .

El primero proporciona funciones hiperbólicas, trigonométricas y logarítmicas para números reales, mientras que el segundo trabaja con números complejos. El tercero admite representaciones exactas de números decimales utilizando aritmética de precisión arbitraria, mientras que el último ofrece soporte para aritmética de números racionales.

2.1 MÓDULO MATH

El módulo *math*, empaquetado en la distribución estándar desde el origen de Python, facilita el acceso a las funciones matemáticas usuales definidas por el estándar C, para operar con números reales y por extensión a números enteros, mientras que para trabajar con números complejos debemos hacer uso del módulo *cmath*.

El módulo *math* consiste, por tanto, en una envoltura simple alrededor de las funciones de la biblioteca de matemáticas de la plataforma C.

Para poder usar los métodos del módulo *math* debemos empezar importando el módulo.

```
1. | import math
```

Vamos a ver las funciones más comunes ofrecidas por este módulo. En la *documentación de Python para math (https://docs.python.org/3/library/math.html)* podéis encontrar todas las funciones disponibles.

2.1.1 Constantes matemáticas

El módulo *math* de Python ofrece una serie de constantes predefinidas que facilitan su uso y proporcionan consistencia en todo el código.

Constante	Descripción
math.*e*	La constante de Euler (2.7182...).
math.*inf*	Un infinito positivo en coma flotante.
math.*nan*	El valor NaN (*Not a Number* – No es un número) en coma flotante.
math.*pi*	La constante PI (3.1415...).
math.*tau*	La constante tau (6.2831...).

2.1.2 Funciones matemáticas

Disponemos de un amplio conjunto de funciones aritméticas, logarítmicas y trigonométricas en *math*.

A menos que se mencione explícitamente todos los valores devueltos por las funciones son en coma flotante (*float*).

Funciones	Descripción
math.*acos*(x)	Devuelve el arcocoseno de **x**, en radianes. El resultado está entre 0 y pi.
math.*acosh*(x)	Devuelve el coseno hiperbólico inverso de **x**.
math.*asin*(x)	Devuelve el arcoseno de **x**, en radianes. El resultado está entre -pi/2 y pi/2
math.*asinh*(x)	Devuelve el seno hiperbólico inverso de **x**.
math.*atan*(x)	Devuelve el arcotangente de **x**, en radianes. El resultado está entre -pi/2 y pi/2.
math.*atan2*(x, y)	Devuelve **atan(y / x)**, en radianes. El resultado está entre -pi y pi.
math.*atanh*(x)	Devuelve la tangente hiperbólica inversa de **x**.
math.*ceil*(x)	Devuelve el valor superior de un número, el menor entero mayor o igual que **x**.
math.*comb*(n, k)	Devuelve el número de **k** elementos de entre **n** elementos sin repetición ni orden. Evalúa **n! / (k! * (n - k)!)** cuando k <= n y se evalúa a cero cuando k > n. Genera un error *TypeError* si alguno de los argumentos no es un número entero. Genera un error *ValueError* si alguno de los argumentos es negativo.

math.*copysign*(x, y)	Devuelve un número en punto flotante con la magnitud (valor absoluto) de **x** pero con el signo de **y**. En las plataformas que soportan ceros con signo, copysign(1.0, -0.0) devuelve -1.0.
math.*cos*(x)	Devuelve el coseno de **x** radianes.
math.*cosh*(x)	Devuelve el coseno hiperbólico de **x**.
math.*degrees*(x)	Convierte el ángulo **x** de radianes a grados.
math.*dist*(p, q)	Devuelve la distancia euclidiana entre dos puntos **p** y **q**, cada uno dado como una secuencia (o iterable) de coordenadas. Los dos puntos deben tener la misma dimensión. Equivale aproximadamente a: **sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))**
math.*erf*(x)	Devuelve la función de error de **x**.
math.*erfc*(x)	Devuelve la función de error complementaria de **x**: **1.0 - erf(x)** Se utiliza para valores grandes de **x** en los que una sustracción de uno causaría una pérdida de precisión.
math.*exp*(x)	Devuelve e elevado a la potencia de **x**. Es más preciso que **math.e ** x** o **pow(math.e, x)**.
math.*expm1*(x)	Devuelve e elevado a la potencia de **x** – 1. Para valores pequeños de **x**, la sustracción en **exp(x) - 1** puede resultar en una pérdida de precisión; esta funciónproporciona una manera de calcularlo con total precisión.
math.*fabs*(x)	Devuelve el valor absoluto de **x**.
math.*factorial*(x)	Devuelve el factorial de un entero **x**. Lanza un *ValueError* si **x** no es entero o es un valor negativo.
math.*floor*(x)	Devuelve el mayor entero menor o igual que **x**.
math.*fmod*(x, y)	Devuelve el resto de **x/y**. La expresión de Python **x%y** puede no devolver el mismo resultado.
math.*frexp*(x)	Devuelve la mantisa y el exponente de **x** como el par **(m, e)**, donde m es un número en punto flotante y e es un entero tal que **x == m * 2**e**. Si x es cero, devuelve (0.0, 0).
math.*fsum*(iterable)	Devuelve una suma exacta en coma flotante de los valores del iterable. Evita la pérdida de precisión mediante la realización de múltiples sumas parciales intermedias.
math.*gamma*(x)	Devuelve la función Gamma en **x**.
math.*gcd*(*integers)	Devuelve el máximo común divisor de los argumentos enteros especificados. Si todos los argumentos son cero, el valor devuelto es 0. Sin argumentos devuelve 0.

math.*hypot*(*coordinates)	Devuelve la norma euclidiana: **sqrt(sum(x**2 for x in coordinates))** Es la longitud del vector desde el origen hasta el punto dado por las coordenadas. Para un punto bidimensional (x, y), esto equivale a calcular la hipotenusa de un triángulo rectángulo utilizando el teorema de Pitágoras: **sqrt(x*x + y*y)**
math.*isclose*(a, b, *, rel_tol=1e-09, abs_tol=0.0)	Devuelve *True* si los valores **a** y **b** están cerca el uno del otro y *False* en caso contrario. El hecho de que dos valores se consideren cercanos se determina según las tolerancias absoluta y relativa dadas. **rel_tol** es la tolerancia relativa. Es la máxima diferencia permitida entre a y b, en relación con el valor absoluto mayor de a o b. **abs_tol** es la tolerancia absoluta mínima. Útil para comparaciones cercanas a cero. Debe ser al menos cero. Si no hay errores, el resultado será: **abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)**
math.*isfinite*(x)	Devuelve *True* si **x** no es un infinito ni un *NaN*, y *False* en caso contrario. 0.0 se considera finito.
math.*isinf*(x)	Devuelve *True* si **x** es un infinito positivo o negativo, y *False* en caso contrario.
math.*isnan*(x)	Devuelve *True* si **x** es un *NaN*, y *False* en caso contrario.
math.*isqrt*(n)	Devuelve la raíz cuadrada entera del entero no negativo **n**. Es equivalente al mayor entero a tal que **a**a <= n**.
math.*lcm*(*integers)	Devuelve el mínimo común múltiplo de los argumentos enteros especificados. Si todos los argumentos son distintos de cero, el valor devuelto es el menor número entero positivo que es múltiplo de todos los argumentos. Si alguno de los argumentos es cero, el valor devuelto es 0. Sin argumentos devuelve 1.
math.*ldexp*(x, i)	Devuelve **x * (2**i)**. Corresponde a la inversa de la función *frexp*().
math.*lgamma*(x)	Devuelve el logaritmo natural del valor absoluto de la función Gamma en **x**.
math.*log*(x [, base])	Devuelve el logaritmo natural de **x** (en base e). Con dos argumentos, devuelve el logaritmo de **x** en la **base** dada, calculado como: **log(x)/log(base)**
math.*log10*(x)	Devuelve el logaritmo en base 10 de **x**. Suele ser más preciso que **log(x, 10)**.
math.*log1p*(x)	Devuelve el logaritmo natural de **1+x** (en base e). El resultado se calcula de forma más precisa para x próximo a cero.

math.*log2*(x)	Devuelve el logaritmo en base 2 de **x**. Suele ser más preciso que **log(x, 2)**.
math.*modf*(x)	Devuelve las partes fraccionaria y entera de **x**. Ambos resultados llevan el signo de x y son en punto flotante.
math.*nextafter*(x, y)	Devuelve el siguiente valor de punto flotante después de **x** hacia **y**. Si x es igual a y, devuelve y.
math.*perm*(n, k=None)	Devuelve el número de formas de elegir **k** elementos de entre los **n** elementos sin repetición y con orden. Se evalúa a **n! / (n - k)!** cuando k <= n y se evalúa a cero cuando k > n. Si no se especifica k o es *None*, entonces k por defecto es n y la función devuelve n! Lanza un *TypeError* si alguno de los argumentos no es un número entero. Lanza un *ValueError* si alguno de los argumentos es negativo.
math.*pow*(x, y)	Devuelve el valor de **x** a la potencia de **y**. Convierte ambos argumentos al tipo de coma flotante.
math.*prod*(iterable, *, start=1)	Devuelve el producto de todos los elementos del iterable. El valor inicial **start** del producto es 1 por defecto. Si el iterable está vacío, devuelve el valor inicial.
math.*radians*(x)	Convierte el valor **x** de grados a radianes.
math.*remainder*(x, y)	Devuelve el resto de **x** con respecto a **y** al estilo IEEE 754. Para x finito e y finito no nulo, es la diferencia **x - n*y**, donde n es el entero más cercano al valor exacto del cociente **x/y**. Si x/y está exactamente entre dos enteros consecutivos, se utiliza el entero par más cercano para n. El resto **r = remainder (x, y)**, por tanto siempre satisface **abs(r) <= 0.5 * abs(y)**.
math.*sin*(x)	Devuelve el seno de **x** radianes.
math.*sinh*(x)	Devuelve el seno hiperbólico de **x**.
math.*sqrt*(x)	Devuelve la raíz cuadrada de **x**. Lanza un *ValueError* cuando x es un valor negativo.
math.*tan*(x)	Devuelve la tangente de **x** radianes.
math.*tanh*(x)	Devuelve la tangente hiperbólica de **x**.
math.*trunc*(x)	Devuelve **x** con la parte fraccionaria eliminada, dejando la parte entera. Es equivalente a *floor*() para x positivo, y a *ceil*() para x negativo.

2.1.3 Uso del módulo math

A continuación vamos a ver unos ejemplos del uso de las funciones matemáticas del modulo *math*.

Manejaremos las constantes matemáticas, así como funciones aritméticas, exponenciales y logarítmicas, además de funciones trigonométricas e hiperbólicas y terminaremos con un apartado de funciones especiales.

2.1.3.1 CONSTANTES

Las principales constantes matemáticas en el módulo *math* de Python son:

- **Pi**. El cociente entre la circunferencia y el diámetro de un círculo.
- **Tau**. La relación entre la circunferencia y el radio de un círculo, con un valor igual a 2*PI.
- El **número de Euler**. La base del logaritmo natural. Se emplea para calcular las tasas de crecimiento de la población a lo largo del tiempo o de desintegración radiactiva.
- **Infinito**. Un concepto matemático que representa algo que no tiene límites y no puede definirse con un número. Lo emplearemos en algoritmos cuando se tenga que comparar un valor con un máximo o mínimo absoluto.
- **NaN**. Es un concepto informático creado para referenciar un valor que no es numérico. Indica que una variable que debería ser numérica contiene un dato que no representa un número.

math_01_constantes.py

```
1.  import math
2.
3.
4.  #constante pi
5.  print('Constante Pi')
6.  print('PI:', math.pi)
7.
8.  radio = 5 # longitud radio
9.  longitud = 2 * math.pi * radio  # perímetro circunferencia
10. area = math.pi * radio * radio  # área circunferencia
11. print(f'Radio: {radio}, Perímetro: {longitud:.8}, Área: {area:.8}')
12.
13. # constante tau
14. print('\nConstante Tau')
15. print('TAU:', math.tau)
16.
17. longitud = math.tau * radio # perímetro circunferencia
18. print(f'Radio: {radio}, Perímetro: {longitud:.8}')
19.
20. # constante de Euler
21. print('\nConstante e')
22. print('EULER:', math.e)
23.
```

```
# infinito
print('\nInfinito')
print('+Infinito:', math.inf)
print('-Infinito:', -math.inf)
print('Infinito:', float("inf"))
x = 1.7976931348623157e+308 # valor máximo de un float
print(f'{x} > math.inf:', x > math.inf)
print(f'{x} > -math.inf:', x > -math.inf)

# valor no numérico
print('\nValor no numérico')
print('NaN:', math.nan)
print('NaN:', float("nan"))
```

En el resultado del guión vemos los valores de PI, TAU y EULER, así como la representación de los valores infinito, tanto positivo como negativo, y el valor no-numérico.

Calculamos el valor de la circunferencia con PI y con TAU y el área con PI.

Y comparamos el valor máximo de un número en coma flotante en Python con infinito positivo y negativo.

```
Constante Pi
PI: 3.141592653589793
Radio: 5, Perímetro: 31.415927, Área: 78.539816

Constante Tau
TAU: 6.283185307179586
Radio: 5, Perímetro: 31.415927

Constante e
EULER: 2.718281828459045

Infinito
+Infinito: inf
-Infinito: -inf
Infinito: inf
1.7976931348623157e+308 > math.inf: False
1.7976931348623157e+308 > -math.inf: True

Valor no numérico
NaN: nan
NaN: nan
```

2.1.3.2 FUNCIONES ARITMÉTICAS

Para obtener valores enteros de un número en punto flotante disponemos de funciones que truncan *trunc*() el número decimal o nos devuelven el techo *ceil*() o el suelo *floor*() del número.

Las operaciones en coma flotante son propensas a errores a lo largo de sucesivos cálculos, debido a la representación numérica en la computadora. En Python disponemos de la función *isclose*(), que establece una tolerancia en las comparaciones con lo que podemos minimizar los errores. La comparación es equivalente a:

```
abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)
```

La diferencia entre los valores debe ser menor o igual a la tolerancia relativa (por defecto 1e-09) multiplicada por el mayor valor absoluto de los valores. La tolerancia absoluta es la diferencia máxima para considerarlo cercano independientemente de la magnitud de los valores de entrada (por defecto 0.0).

La función *isclose*() devuelve *True* si dos números están dentro de la tolerancia establecida, en caso contrario devuelve *False*.

También podemos minimizar errores en operaciones repetitivas mediante el uso de *fsum*(), que realiza múltiples sumas parciales y es más precisa que la función integrada *sum*().

Python dispone del operador porcentaje (%) para calcular el módulo de la división, que funciona bien con enteros, pero puede perder precisión con números reales. La función *fmod*() nos soluciona este inconveniente.

Los cálculos del máximo común divisor *gcd*() y mínimo común múltiplo *lcm*(), este último desde la versión de Python 3.9, nos ofrecen los resultados correspondientes para la serie de valores que proporcionemos a las funciones.

El factorial de un número con *factorial*() y el cálculo de la función gamma con *gamma*(), que extiende el concepto de factorial a los números reales, también están contemplados en el módulo *math*.

math_02_aritmetica.py

```
1.  import math
2.
3.
4.  # conversión a enteros
5.  print('Conversión a enteros')
6.  print('Valor entero trunca   techo suelo')
7.  TEST = [-5.3, -1.5, -0.2, 0.2, 1.5, 5.3]
8.  for x in TEST:
9.  i = int(x)
```

```
t = math.trunc(x)
c = math.ceil(x)
f = math.floor(x)
print(f'{x:>+5.2f}  {i:>6d}  {t:>6d}  {c:>6d} {f:>6d}')

# compración de números
print('\nComparación isclose(5.999, 6)')
print('Comparación sin tolerancia:', math.isclose(5.999, 6))
print('Comparación tolerancia relativa:', math.isclose(5.999, 6, rel_tol=0.1))
print('\nComparación isclose(1, 1.00001)')
print('Comparación sin tolerancia:', math.isclose(1, 1.00001))
print('Comparación tolerancia absoluta:', math.isclose(1, 1.00001, abs_tol=1e-4))

# minimizar errores en operaciones repetitivas
print('\nMinimizar errores en suma')
TEST = [.1] * 10
print('Valores:', TEST)
n = sum(TEST)
print(f'Con sum: {n}')
m = math.fsum(TEST)
print(f'Con fsum: {m}')

# módulo de la división (ojo con el signo)
print('\nMódulo de la división')
TEST = [(5, 2), (5, -2), (-5, 2)]
print(f'{"x":>4s}  {"y":>4s}{"%":>7s} {"fmod":>7s}')
for x, y in TEST:
modulo = x % y
fmodulo = math.fmod(x, y)
print(f'{x:4.1f}  {y:4.1f}  {modulo:5.2f} {fmodulo:5.2f}')

# máximo común divisor
mcd = math.gcd(10,125)
print(f'\nMCD(10,125): {mcd}')
# mínimo común múltiplo
mcm = math.lcm(10,125)
print(f'MCM(10,125): {mcm}')

# factorial
print('\nFactorial')
for i in range(6):
print(f'{i}! :', math.factorial(i))

# gamma y lgamma
print(f'\nValor{"gamma":>22s}{"lgamma":>22s}{"logaritmo":>22s}')
for i in range(1, 6):
x = i * 1.1
g = math.gamma(x)
lg = math.lgamma(x)
```

```
59.  logg = math.log(math.gamma(x))
60.  print(f'{x:>5.2f}  {g:>20.17f}  {lg:>20.17f}  {logg:>20.17f}')
61.
62.  # partes fraccionaria y entera
63.  print(f'\nValor{" ":22s}Partes fraccionaria y entera')
64.  for i in range(7):
65.  x = i / 3.0
66.  print(f'{i}/3 :{x:>20.17f} ', math.modf(x))
```

A continuación, en los resultados obtenidos de la ejecución del guión, vemos los distintos valores de la conversión a enteros con las funciones de truncamiento, techo y suelo. Las comparaciones con *isclose*(). Y el resultado de una suma repetitiva con una precisión mayor cuando usamos *fsum*().

Debemos prestar atención al uso de *fmod*() para calcular el módulo de la división, ya que cambia el signo del resultado, como podemos observar.

Seguimos con los cálculos de máximo común divisor y mínimo común múltiplo. El factorial y la función gamma. Y por último la representación de las partes fraccionaria y entera de diversos números.

```
Conversión a enteros
   Valor  entero  trunca   techo   suelo
   -5.30      -5      -5      -5      -6
   -1.50      -1      -1      -1      -2
   -0.20       0       0       0      -1
   +0.20       0       0       1       0
   +1.50       1       1       2       1
   +5.30       5       5       6       5

Comparación isclose(5.999, 6) Comparación sin tolerancia: False Comparación to-
lerancia relativa: True

Comparación isclose(1, 1.00001) Comparación sin tolerancia: False Comparación
tolerancia absoluta: True

Minimizar errores en suma
Valores: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,0.1]
Con sum: 0.9999999999999999
Con fsum: 1.0

Módulo de la división
      x       y       %    fmod
    5.0     2.0    1.00    1.00
    5.0    -2.0   -1.00    1.00
   -5.0     2.0    1.00   -1.00
```

```
MCD(10,125):5
MCM(10,125):250

Factorial
0! :1
1! :1
2! :2
3! :6
4! :24
5! :120
```

Valor	gamma	lgamma	logaritmo
1.10	0.95135076986687295	-0.04987244125984036	-0.04987244125983997
2.20	1.10180249087971260	0.09694746679063826	0.09694746679063866
3.30	2.68343738195576886	0.98709857789473476	0.98709857789473465
4.40	10.13610185115513396	2.31610349142485727	2.31610349142485727
5.50	52.34277778455351893	3.95781396761871651	3.95781396761871607

Valor	Partes fraccionaria y entera
0/3 : 0.00000000000000000	(0.0, 0.0)
1/3 : 0.33333333333333331	(0.3333333333333333, 0.0)
2/3 : 0.66666666666666663	(0.6666666666666666, 0.0)
3/3 : 1.00000000000000000	(0.0, 1.0)
4/3 : 1.33333333333333326	(0.33333333333333326, 1.0)
5/3 : 1.66666666666666674	(0.6666666666666667, 1.0)
6/3 : 2.00000000000000000	(0.0, 2.0)

2.1.3.3 FUNCIONES EXPONENCIALES Y LOGARÍTMICAS

En Python las potencias de un número se pueden calcular con el operador doble asterisco (**), la función integrada *pow*() o la función del módulo *math* del mismo nombre, *math.pow*(), que convierte los argumentos a tipo *float* antes de realizar el cálculo. Aunque los tres métodos hacen lo mismo, existen diferencias en cuanto a la implementación que se reflejan en los tiempos de ejecución, donde *math.pow*() es más rápida que los otros métodos. Sin embargo, no es posible calcular raíces cuadradas de números negativos con *math*, ya que esto requiere el uso de números complejos.

Si el exponente de la potencia es inferior a 1, *pow*() calcula la raíz correspondiente.

El exponente natural de un número lo calculamos en *math* con la función *exp*(). También podemos calcularlo haciendo uso del operador (**) o de la función integrada *pow*(), pero al igual que comentábamos más arriba, la función *math.exp*() es más rápida que cualquiera de los otros métodos.

El logaritmo es la función inversa a la exponenciación. El logaritmo en base b de un número real positivo n, es el exponente x al que hay que elevar b para obtener n.

```
n = b ** x
```

Los logaritmos de números menores que 1 producen resultados negativos.

A causa de los errores de redondeo en coma flotante, el valor producido por *log*(x, base) tiene una precisión limitada. Es recomendable emplear *log10*() o *log2*(), para logaritmos en bases 10 o 2, que proporcionan valores con mayor precisión.

Igualmente *log1p*() es más preciso para valores muy cercanos a cero que *log*(1+x).

math_03_exp_log.py

```
import math


TEST = [(3, 2), (4, 0), (16, 1), (9, 0.5)]

# exponenciación
print('Potencias')
print(f'{"x":>5s} {"y":>5s}{"**":>7s} {"pow":>7s}')
for x, y in TEST:
xy = x ** y
xypow = math.pow(x, y)
print(f'{x:5.2f}**{y:4.2f} {xy:6.3f}  {xy:6.3f}')

print('\nPotencias de e')
print(f'{"x":>5s} {"exp":>8s} {"pow":>8s} {"expm1":>8s} {"exp - 1":>8s}')
for x in range(1, 6):
ex = math.exp(x)
expw = math.pow(math.e, x)
exp1 = math.expm1(x)
expsub = math.exp(x) - 1
print(f'{x:5.3f}  {ex:7.3f}  {expw:7.3f}  {exp1:7.3f}  {expsub:7.3f}')

# raíces
print('\nRaíces')
print(f'{"x":>5s} {"sqrt":>6s} {"pow":>6s}')
for x, _ in TEST:
y = math.sqrt(x)
z = math.pow(x, 0.5)
print(f'{x:5.2f} {y:6.3f} {z:6.3f}')


# logaritmos
print('\nLogaritmos')
print(f'{"x":>5s} {"log":>6s} {"log1p":>6s} {"log 2":>6s} {"log2":>6s} ',
```

```
35. f'{"log 10":>6s} {"log10":>6s}')
36. for i in range(1, 6):
37. lge = math.log(i)
38. lg1p = math.log1p(i)
39. lg2 = math.log(i, 2)
40. log2 = math.log2(i)
41. lg10 = math.log(i, 10)
42. log10 = math.log10(i)
43. print(f'{i:5.3f}  {lge:5.3f}  {lg1p:5.3f} ',
44. f'{lg2:5.3f}  {log2:5.3f}  ',
45. f'{lg10:5.3f}  {log10:5.3f}')
46.
47. # Logaritmos de números cercanos a cero
48. print('\nLogaritmos de números cercanos a 0')
49. print('log(0.0000025)  ', math.log(0.0000025))
50. print('log1p(0.0000025)   ', math.log1p(0.0000025))
51. print('log(1.0000025)  ', math.log(1.0000025))
52. print('log(0.0000000025)  ', math.log(0.0000000025))
53. print('log1p(0.0000000025)', math.log1p(0.0000000025))
54. print('log(1.0000000025)  ', math.log(1.0000000025))
```

En el resultado del guión vemos los diversos valores obtenidos del cálculo de potencias, raíces y logaritmos.

```
Potencias
     x     y      **     pow
  3.00**2.00   9.000   9.000
  4.00**0.00   1.000   1.000
 16.00**1.00  16.000  16.000
  9.00**0.50   3.000   3.000

Potencias de e
     x      exp      pow    expml  exp - 1
 1.000    2.718    2.718    1.718    1.718
 2.000    7.389    7.389    6.389    6.389
 3.000   20.086   20.086   19.086   19.086
 4.000   54.598   54.598   53.598   53.598
 5.000  148.413  148.413  147.413  147.413

Raíces
     x    sqrt     pow
  3.00   1.732   1.732
  4.00   2.000   2.000
 16.00   4.000   4.000
  9.00   3.000   3.000
```

```
Logaritmos
     x     log   log1p    log2    log2  log 10   log10
 1.000   0.000   0.693   0.000   0.000   0.000   0.000
 2.000   0.693   1.099   1.000   1.000   0.301   0.301
 3.000   1.099   1.386   1.585   1.585   0.477   0.477
 4.000   1.386   1.609   2.000   2.000   0.602   0.602
 5.000   1.609   1.792   2.322   2.322   0.699   0.699

Logaritmos de números cercanos a 0
log(0.0000025)            -12.89921982609012
log1p(0.0000025)          2.4999968750052084e-06
log(1.0000025)            2.4999968749105643e-06
log(0.0000000025)         -19.806975105072254
log1p(0.0000000025)       2.499999996875e-09
log(1.0000000025)         2.4999999816813227e-09
```

2.1.3.4 FUNCIONES TRIGONOMÉTRICAS E HIPERBÓLICAS

La trigonometría es una rama de las matemáticas dedicada al estudio de la relación entre los lados y ángulos de un triángulo rectángulo y una circunferencia.

- El **seno** es la razón entre el cateto opuesto al ángulo y la hipotenusa.
- El **coseno** es la razón entre el cateto adyacente o contiguo al ángulo y la hipotenusa.
- La **tangente** es la razón entre el cateto opuesto al ángulo y el cateto adyacente.
- La **cosecante** es la razón inversa de seno La **secante** es la razón inversa de coseno
- La **cotangente** es la razón inversa de la tangente

Las funciones hiperbólicas son análogas a las funciones trigonométricas ordinarias, pero definidas utilizando la hipérbola en lugar del círculo.

Todas las funciones trigonométricas en el módulo *math* reciben los ángulos expresados en **radianes**, por lo que empezaremos viendo las funciones trigonométricas con las conversiones grados-radianes-grados.

El módulo *math* dispone de la función *hypot*() para calcular la longitud de la hipotenusa dados los catetos del triangulo rectángulo.

Antes de Python 3.8, la función *sqrt*() se empleaba para encontrar la hipotenusa de un triángulo rectángulo:

```
sqrt(x*x + y*y)
```

A partir de Python 3.8, este método se utiliza también para calcular la norma euclidiana. Para los casos de n dimensiones donde las coordenadas se proporcionan de la forma (x1, x2, x3, ..., xn), la longitud euclidiana desde el origen se calcula de la forma:

```
sqrt(x1*x1 + x2*x2 + x3*x3 +....+ xn*xn)
```

A partir de Python 3.8 podemos calcular la distancia euclidiana entre dos puntos con la función *dist*(), que recibe los puntos como un iterable de coordenadas. Veremos el cálculo con la función *hypot*() y con *dist*().

math_04_trigonometria.py

```
import math


# conversión grados en radianes
# radianes = grados * PI / 180
print('Grados a radianes')
print(f'{"grados":>5s}{"radianes":>9s}{"grados":>8s}')
for grados in range(0, 361, 30):
radianes = math.radians(grados)
grados2 = math.degrees(radianes)
print(f'{grados:4d}{radianes:7.5f} {grados2:3.0f}')

# valores en el primer cuadrante
print('\nTrigonometría')
print(f'{"grados":>5s}{"radianes":>9s}',
f'{"seno":>8s} {"coseno":>8s}  {"tangente":>8s}')
for grados in [0, 30, 45, 60, 90]:
radianes = math.radians(grados)
seno = math.sin(radianes)
coseno = math.cos(radianes)
# tangente de 90 grados (PI/2) no está definida
if grados == 90:
tangente = math.inf
else:
tangente = math.tan(radianes)
print(f'{grados:4d}{radianes:7.5f} ',
f'{seno:7.5f}  {coseno:7.5f} {tangente:7.5f}')

# funciones trigonométricas inversas
print('\nFunciones trigonométricas inversas')
print(f'{"radianes":>5s}',
f'{"arcoseno":>8s} {"arcocoseno":>8s} {"arcotangente":>8s}')
for radianes in [0., .25, .5, .75, 1.]:
aseno = math.asin(radianes)
acoseno = math.acos(radianes)
atangente = math.atan(radianes)
print(f' {radianes:7.5f} ',
f'{aseno:7.5f}  {acoseno:7.5f}  {atangente:7.5f}')
```

```
39.
40.  # funciones hiperbólicas
41.  print('\nFunciones hiperbólicas')
42.  print(f'{"grados":>5s}{"radianes":>9s}',
43.  f'{"senoh":>8s} {"cosenoh":>8s}  {"tangenteh":>8s}')
44.  for grados in [0, 30, 45, 60, 90]:
45.  radianes = math.radians(grados)
46.  senoh = math.sinh(radianes)
47.  cosenoh = math.cosh(radianes)
48.  tangenteh = math.tanh(radianes)
49.  print(f'{grados:4d}{radianes:7.5f} ',
50.  f'{senoh:7.5f}  {cosenoh:7.5f}  {tangenteh:7.5f}')
51.
52.  # cálculo hipotenusa
53.  print('\nCálculo de la hipotenusa')
54.  TEST = [(1, 1), (3, 4), (math.pi/4, math.pi/4)]
55.  print(f'{"x":>5s} {"y":>5s}{"hipotenusa":>12s}')
56.  for x, y in TEST:
57.  h = math.hypot(x, y)
58.  print(f'{x:5.3f} {y:5.3f} {h:7.5f}')
59.
60.  # distancia entre dos puntos
61.  print('\nCálculo de la distancia entre dos puntos')
62.  TEST = [((0, 0), (3, 4)), ((0, 3), (4, 0)),
63.  ((-3, -4), (0, 0)), ((1, 1), (0, 0))]
64.  print(f'{"punto1":>8s}  {"punto2":>8s}{"hypot":>9s}{"dist":>9s}')
65.  for (x1, y1), (x2, y2) in TEST:
66.  x = x1 - x2
67.  y = y1 - y2
68.  h = math.hypot(x, y)
69.  d = math.dist((x1, y1), (x2, y2))
70.  print(f'({x1:3d},{y1:3d}) ({x2:3d},{y2:3d}) {h:7.5f}  {d:7.5f}')
```

Vemos a continuación los resultados de la ejecución del guión.

```
Grados a radianes
 grados  radianes  grados
      0   0.00000       0
     30   0.52360      30
     60   1.04720      60
     90   1.57080      90
    120   2.09440     120
    150   2.61799     150
    180   3.14159     180
    210   3.66519     210
    240   4.18879     240
    270   4.71239     270
    300   5.23599     300
    330   5.75959     330
    360   6.28319     360
```

```
Trigonometría
 grados  radianes      seno    coseno  tangente
      0   0.00000   0.00000   1.00000   0.00000
     30   0.52360   0.50000   0.86603   0.57735
     45   0.78540   0.70711   0.70711   1.00000
     60   1.04720   0.86603   0.50000   1.73205
     90   1.57080   1.00000   0.00000       inf

Funciones trigonométricas inversas
 radianes  arcoseno  arcocoseno  arcotangente
  0.00000   0.00000     1.57080       0.00000
  0.25000   0.25268     1.31812       0.24498
  0.50000   0.52360     1.04720       0.46365
  0.75000   0.84806     0.72273       0.64350
  1.00000   1.57080     0.00000       0.78540

Funciones hiperbólicas
 grados  radianes    senoh  cosenoh  tangenteh
      0   0.00000  0.00000  1.00000    0.00000
     30   0.52360  0.54785  1.14024    0.48047
     45   0.78540  0.86867  1.32461    0.65579
     60   1.04720  1.24937  1.60029    0.78071
     90   1.57080  2.30130  2.50918    0.91715

Cálculo de la hipotenusa
     x      y   hipotenusa
 1.000  1.000      1.41421
 3.000  4.000      5.00000
 0.785  0.785      1.11072

Cálculo de la distancia entre dos puntos
  punto1    punto2     hypot      dist
   (0,0)     (3,4)   5.00000   5.00000
   (0,3)     (4,0)   5.00000   5.00000
 (-3,-4)     (0,0)   5.00000   5.00000
   (1,1)     (0,0)   1.41421   1.41421
```

2.1.3.5 FUNCIONES ESPECIALES

El módulo *math* ofrece las funciones para cálculo combinatorio *comb*(), que devuelve el número de maneras de elegir elementos sin repetición de un conjunto sin un orden particular, y *perm*(), que devuelve el número de maneras de elegir elementos sin repetición y con un orden.

Podemos obtener la raíz cuadrada entera de un número con *isqrt*(). Y calcular el producto de todos los elementos de un iterable con *prod*().

La función de error *erf*() puede utilizarse para calcular funciones estadísticas tradicionales como la **distribución normal acumulativa**:

```
def phi(x):
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

O la **función de error complementaria** *erfc*(x), que devuelve **1.0 - erf(x)** y que para valores grandes ofrece resultados más precisos.

math_05_especiales.py

```
import math


# combinatoria
print('Combinatoria')
n = 10
k = 2
combinaciones = math.comb(n, k)
permutaciones = math.perm(n, k)
print(f'Elementos {n} tomados de a {k}')
print('Combinaciones:', combinaciones)
print('Permutaciones:', permutaciones)

# raíz cuadrada entera
print('\nRaíz cuadrada entera')
print(f'{"x":>3s}{"isqrt":>7s}{"sqrt":>8s}')
for x in range(25, 110, 25):
i = math.isqrt(x)
f = math.sqrt(x)
print(f'{x:3d}  {i:3d} {f:8.5f}')

# producto de un iterable
print('\nProducto')
TEST = [.5, 2, 3, 4, 5]
producto = math.prod(TEST)
print('Valores:', TEST)
print('Producto:', producto)

# función de error
print('\nFunción de error')
print(f'{"x":>5s} {"erf":>9s}{"erfc":>10s}')
for x in range(-2, 3):
errf = math.erf(x)
errfc = math.erfc(x)
print(f'{x:5.2f}  {errf:8.5f}  {errfc:8.5f}')
```

En el resultado del script vemos las combinaciones y permutaciones de 10 elementos tomados de dos en dos.

La diferencia entre valores de raíces cuadradas enteras y raíces cuadradas decimales. El valor de la multiplicación de una lista de números reales y enteros. Y el cálculo de la función de error y la complementaria para una serie de valores positivos y negativos.

```
Combinatoria
Elementos 10 tomados de a 2
Combinaciones: 45
Permutaciones: 90

Raíz cuadrada entera
   x   isqrt      sqrt
  25       5   5.00000
  50       7   7.07107
  75       8   8.66025
 100      10  10.00000


Producto
Valores: [0.5, 2, 3, 4, 5]
Producto: 60.0

Función de error
      x       erf      erfc
  -2.00  -0.99532   1.99532
  -1.00  -0.84270   1.84270
   0.00   0.00000   1.00000
   1.00   0.84270   0.15730
   2.00   0.99532   0.00468
```

2.2 MÓDULO CMATH

El módulo *math* de Python se complementa con el módulo *cmath*, que proporciona funciones para tareas matemáticas con números complejos.

Un número imaginario o complejo se representa como la suma de un número real y un número imaginario de la forma **a** + **bi**, donde **a** es el número real y **bi** es el número imaginario, siendo **i** la raíz cuadrada de -1.

Los números complejos o imaginarios tienen su uso en muchas aplicaciones relacionadas con las matemáticas y Python proporciona herramientas útiles para manejarlos y manipularlos.

Las funciones de este módulo aceptan números enteros, en coma flotante y complejos. Todas las funciones de *cmath* siempre devuelven un número complejo, incluso si el valor de retorno se puede expresar como un número real, en cuyo caso el número complejo tiene una parte imaginaria de cero.

El motivo de que existan dos módulos distintos es evidente, cuando se utilizan únicamente números reales se desea ver errores sí se intenta calcular la raíz cuadrada de un número negativo, porque realmente es imposible. Por el contrario, cuando se utilizan números complejos dicha tarea resulta natural.

Para usar los métodos del módulo *cmath* debemos empezar importando el módulo.

```
1. | import cmath
```

Vamos a ver las funciones más comunes ofrecidas por este módulo. En la *documentación de Python para cmath (https://docs.python.org/3/library/cmath.html)* podéis encontrar todas las funciones disponibles.

2.2.1 Constantes matemáticas

El módulo *cmath* de Python ofrece una serie de constantes predefinidas, que facilitan su uso y proporcionan consistencia en todo el código.

Constante	Descripción
cmath.*e*	La constante de Euler (2.7182...) en coma flotante.
cmath.*inf*	Devuelve un infinito positivo en coma flotante. Equivale a **float('inf')**.
cmath.*infj*	Número complejo con parte real cero y parte imaginaria infinita positiva. Equivale a **complex(0.0, float('inf'))**.
cmath.*nan*	El valor NaN (*Not a Number* – No es un número) en coma flotante.
cmath.*nanj*	Número complejo con parte real cero y parte imaginaria NaN. Equivale a **complex(0.0, float('nan'))**.
cmath.*pi*	La constante PI (3.1415...) en coma flotante.
cmath.*tau*	La constante tau (6.2831...) en coma flotante.

2.2.2 Funciones matemáticas

El módulo *cmath* ofrece las siguientes funciones.

Funciones	Descripción
cmath.*acos*(x)	Devuelve el arcocoseno de un número complejo.
cmath.*acosh*(x)	Devuelve el coseno hiperbólico inverso de un número complejo.
cmath.*asin*(x)	Devuelve el arcoseno de un número complejo.
cmath.*asinh*(x)	Devuelve el seno hiperbólico inverso de un número complejo.
cmath.*atan*(x)	Devuelve el arcotangente de un número complejo.
cmath.*atanh*(x)	Devuelve la tangente hiperbólica inversa de un número complejo.
cmath.*cos*(x)	Devuelve el coseno de un número complejo.
cmath.*cosh*(x)	Devuelve el coseno hiperbólico de un número complejo.
cmath.*exp*(x)	Devuelve e elevado a la potencia de un número complejo.
cmath.*isclose*(a, b, *, rel_tol=1e-09, abs_tol=0.0)	Devuelve *True* si los valores complejos **a** y **b** están cerca el uno del otro y *False* en caso contrario. El hecho de que dos valores se consideren cercanos se determina según las tolerancias absoluta y relativa dadas. **rel_tol** es la tolerancia relativa - es la máxima diferencia permitida entre a y b, en relación con el valor absoluto mayor de a o b. **abs_tol** es la tolerancia absoluta mínima - útil para comparaciones cercanas a cero. **abs_tol** debe ser al menos cero. Si no hay errores, el resultado será: **abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)**
cmath.*isfinite*(x)	Devuelve *True* si el valor complejo **x** no es un infinito ni un NaN, y *False* en caso contrario.
cmath.*isinf*(x)	Devuelve *True* si el valor complejo **x** es un infinito positivo o negativo, y *False* en caso contrario.
cmath.*isnan*(x)	Devuelve *True* si el valor complejo **x** es un NaN, y *False* en caso contrario.
cmath.*log*(x [, base])	Devuelve el logaritmo natural del número complejo **x** (en base e). Con dos argumentos, devuelve el logaritmo de **x** en la **base** dada.
cmath.*log10*(x)	Devuelve el logaritmo en base 10 del número complejo **x**. Suele ser más preciso que log(x, 10).
cmath.*phase*(x)	Devuelve la fase del número complejo **x**.
cmath.*polar*(x)	Devuelve el número complejo **x** en forma de coordenadas polares. Devuelve una tupla de módulo y fase.

cmath.*rect*(x)	Devuelve las coordenadas polares **x** a la forma rectangular del número complejo. Crea un número complejo con fase y módulo. Este método equivale a: **r * (math.cos(phi) + math.sin(phi)*1j)**. El radio r es la longitud del vector, y phi (ángulo de fase) es el ángulo formado con el eje real.
cmath.*sin*(x)	Devuelve el seno de un número complejo.
cmath.*sinh*(x)	Devuelve el seno hiperbólico de un número complejo.
cmath.*sqrt*(x)	Devuelve la raíz cuadrada de un número complejo.
cmath.*tan*(x)	Devuelve la tangente de un número complejo.
cmath.*tanh*(x)	Devuelve la tangente hiperbólica de un número complejo.

2.2.3 Uso del módulo cmath

A continuación vamos a ver unos ejemplos del uso de funciones matemáticas del modulo *cmath*.

2.2.3.1 CREACIÓN DE NÚMEROS COMPLEJOS

En matemáticas, la unidad imaginaria se suele denotar con **i**, en Python se utiliza **j**, como en física, para denotar los números imaginarios.

Python también proporciona la función incorporada *complex*() que permite crear números complejos.

El módulo de un número complejo es el número real positivo que mide su tamaño y generaliza el valor absoluto de un número real. Se puede acceder a la parte real y la imaginaria mediante los atributos *real* e *imag* respectivamente.

Podemos emplear los operadores matemáticos básicos: suma, resta, multiplicación y división (+, -, *, /) con los números complejos.

cmath_01_creacion.py

```
 1. import cmath
 2.
 3.
 4. # creación de números complejos y sus partes
 5. print('Creación por asignación')
 6. z = 1 + 2j
 7. print('Número complejo:', z) # número complejo
 8. print('Parte real:', z.real) # atributo real
 9. print('Parte imaginaria:', z.imag)  # atributo imaginario
10.
11. print('Creación con la clase complex')
12. y = complex(3, 4)
```

```
print('Número complejo:', y)
print('Parte real:', y.real)
print('Parte imaginaria:', y.imag)

# operaciones básicas
print('\nOperaciones matemáticas')
print(f'z: {z}, y: {y}')
suma = z + y
resta = z - y
multiplicacion = z * y
division = z / y
print('Suma x+y:', suma)
print('Resta z-y:', resta)
print('Multiplicación z*y:', multiplicacion)
print('División z/y:', division)
```

Vemos a continuación el resultado de la ejecución del guión.

```
Creación por asignación
Número complejo: (1+2j)
Parte real: 1.0
Parte imaginaria: 2.0
Creación con la clase complex
Número complejo: (3+4j)
Parte real: 3.0
Parte imaginaria: 4.0

Operaciones matemáticas
z: (1+2j), y: (3+4j)
Suma x+y: (4+6j)
Resta z-y: (-2-2j)
Multiplicación z*y: (-5+10j)
División z/y: (0.44+0.08j)
```

2.2.3.2 COORDENADAS

Un número complejo se almacena internamente usando coordenadas rectangulares o cartesianas. Así, un número complejo z se representa como z = a + bj, donde a representa la parte real y b la parte imaginaria.

Otra forma de describir un número complejo es mediante coordenadas polares, donde el número complejo z se define mediante una tupla del módulo r y el ángulo de fase phi. El módulo r es la distancia de z al origen, mientras que la fase phi es el ángulo contrario a las agujas del reloj, medido en radianes, desde el eje x positivo al segmento de línea

que une el origen con z. La conversión a polar se realiza mediante la función *polar*(), que devuelve un par(r, phi).

El rango de fase va de **–cmath.pi** a **+cmath.pi**. El signo del resultado es el mismo que el de **z.imag**.

El módulo se puede obtener utilizando la función incorporada *abs*() y la fase usando la función *cmath.phase*(), que toma un número complejo como entrada y devuelve un número de punto flotante que representa la fase del número complejo. Es equivalente a:

```
fase = cmath.phase(z) = math.atan2(z.imag, z.real)
```

Podemos utilizar la función *cmath.rect*() para convertir un número complejo en forma polar a forma rectangular pasando el módulo r y la fase phi como argumentos.

Equivale a

```
r * (math.cos(phi) + math.sin(phi) * 1j)
```

Veamos un ejemplo.

cmath_02_coordenadas.py

```
import cmath


# creación de un número complejo
z = 1 + 2j
print('Número complejo:', z)

# módulo
r = abs(z)
print('Módulo:', r)
# fase
phi = cmath.phase(z)
print('Fase:', phi)

# coordenadas polares
polar = cmath.polar(z)
print('\nCoordenadas polares:', polar)

# formato rectangular
rectangular = cmath.rect(r, phi)
print('\nRectangular:', rectangular)
```

El resultado del script es:

```
Número complejo: (1+2j)
Módulo: 2.23606797749979
Fase: 1.1071487177940904

Coordenadas polares: (2.23606797749979, 1.1071487177940904)

Rectangular: (1.0000000000000002+2j)
```

2.3 MÓDULO DECIMAL

Python dispone de números de punto flotante basados en los tipos *float* y *double* de C. Sin embargo, los números de punto flotante no representan ciertas fracciones decimales con precisión. Con el módulo *decimal* se puede representar estas fracciones hasta un límite de precisión especificado por el usuario.

Y como dice la Especificación General de Aritmética Decimal (*General Decimal Arithmetic Specification (https://speleotrove.com/decimal/decarith.html)*):

> Decimal "se basa en un modelo de coma flotante que se diseñó pensando en las personas, y tiene necesariamente un principio rector primordial: los ordenadores deben proporcionar una aritmética que funcione de la misma manera que la aritmética que las personas aprenden en la escuela".

Así pues, el propósito del módulo *decimal* es dar soporte a la aritmética utilizando reglas familiares y evitar problemas de representación asociados al punto flotante. Una instancia de *decimal* puede representar cualquier número con exactitud, redondear hacia arriba o hacia abajo y aplicar un límite al número de dígitos significativos. Ofrece soporte a la aritmética decimal de punto flotante rápida y correctamente redondeada, independiente de la máquina. El módulo *decimal* representa los números decimales con una precisión de hasta 28 dígitos decimales, mientras que los números de punto flotante sólo tienen una precisión de hasta 18 dígitos.

El diseño del módulo se centra en tres conceptos: el número decimal, el contexto para la aritmética y las señales.

- Un **número decimal** es inmutable. Tiene un signo, dígitos de coeficiente y un exponente.
- El **contexto** de la aritmética es un entorno que especifica la precisión, las reglas de redondeo, los límites de los exponentes, los indicadores de los resultados de las operaciones. Los contextos pueden aplicarse a todas las instancias de decimal en un hilo o localmente dentro de una pequeña región de código.

- Las **señales** son grupos de condiciones excepcionales que surgen durante el curso del cálculo. Dependiendo de las necesidades de la aplicación las señales pueden ignorarse, considerarse como informativas o tratarse como excepciones.

Es importante utilizar decimales cuando la precisión sea primordial, como en los cálculos financieros. Los decimales pueden sufrir sus propios problemas de precisión, pero en general, los decimales son más precisos que los flotantes.

Para usar los métodos del módulo *decimal* debemos empezar importando el módulo.

```
1. | import decimal
```

Vamos a ver las funciones más importantes ofrecidas por este módulo. En la *documentación de Python para decimal (https://docs.python.org/3/library/decimal.html)* podéis encontrar todas las funciones disponibles.

El módulo *decimal* contiene dos clases, *Decimal*() y *Context*(). Las instancias de *Decimal*() representan números, mientras que las instancias de *Context*() encapsulan diversos ajustes para las operaciones decimales.

2.3.1 Decimal

Un objeto decimal puede construirse de varias maneras, con la clase *Decimal*(), a partir de un par de enteros, de otro número racional, de una cadena o con una tupla.

Clase	Descripción
decimal.*Decimal*(value='0', context=None)	Construye un nuevo objeto decimal. **value** puede ser un entero, una cadena, una tupla, un punto flotante u otro objeto Decimal. Si no se da ningún valor, devuelve *Decimal*('0'). Si es una **cadena**, debe ajustarse a la sintaxis numérica decimal después de eliminar los espacios en blanco iniciales y finales. Si el valor es una **tupla**, debe tener tres componentes, un signo (0 para positivo o 1 para negativo), una tupla de dígitos y un exponente entero. Si es un valor en **punto flotante**, el valor binario en coma flotante se convierte sin pérdida a su equivalente decimal exacto. Esta conversión puede requerir a menudo 53 o más dígitos de precisión. La precisión del contexto no afecta al número de dígitos que se almacenan. Eso se determina exclusivamente por el número de dígitos del valor. El argumento de contexto **context** determina qué hacer si el valor es una cadena malformada. Si el contexto atrapa *InvalidOperation*, se lanza una excepción; en caso contrario, el constructor devuelve un nuevo decimal con el valor *NaN*.

Se dispone de un constructor alternativo mediante la función *from_float*(), que devuelve una instancia de *Decimal*() a partir de un número en punto flotante o un entero. O con los métodos de la clase *Context*() *create_decimal*() y *create_decimal_from_float*().

2.3.2 Métodos de Decimal

Los objetos de *Decimal*() comparten propiedades con los otros tipos numéricos incorporados en Python, como *float* e *int*, además de disponer de todas las operaciones matemáticas y métodos especiales habituales.

Existen algunas pequeñas diferencias entre la aritmética sobre objetos decimales y la aritmética sobre enteros y punto flotante. Cuando se aplica el operador **%** a objetos decimales, el signo del resultado es el signo del dividendo en lugar del signo del divisor.

El operador de división de enteros (//) se comporta de forma análoga, devolviendo la parte entera del cociente truncando hacia cero en lugar de su mínimo, para preservar la identidad habitual

```
x == (x // y) * y + x % y
```

Además de las propiedades numéricas estándar, los objetos decimales de punto flotante también disponen de una serie de métodos especializados.

Método	Descripción
d.*as_integer_ratio*()	Devuelve un par **(n, d)** de enteros que representan la instancia decimal dada como una fracción, en términos menores y con denominador positivo.
d.*as_tuple*()	Devuelve una representación de tupla con nombre del número: **(signo, dígitos, exponente)**.
d.*compare*(other, context=None)	Compara los valores de dos decimales. Devuelve una instancia de *Decimal*(). La comparación se efectúa restando el segundo operando del primero y devolviendo un valor según el resultado de la resta: -1 si el resultado es menor que cero. si el resultado es cero o cero negativo. si el resultado es mayor que cero. Así: -1 => a < b 0 => a == b 1 => a > b Si cualquiera de los operandos es un *NaN* entonces el resultado es un *NaN*.

d.*compare_total*(other, context=None)	Compara los dos operandos utilizando su representación abstracta. No es como *compare*(), que utiliza su valor numérico. Se define una ordenación total para todas las representaciones abstractas posibles.
d.*compare_total_mag*(other, context=None)	Compara los dos operandos utilizando su representación abstracta ignorando el signo.
d.*copy_abs*()	Devuelve una copia sin signo.
d.*copy_negate*()	Devuelve una copia con el signo invertido.
d.*copy_sign*(other, context=None)	Devuelve una copia con el signo igual al signo del operando.
d.*exp*(context=None)	Devuelve e elevado a la potencia del operando.
d.*fma*(other, third, context=None)	Devuelve el operando multiplicado por **other**, más **third**. Se realiza una multiplicación y suma sin redondeo intermedio, solo se redondea al final. **fma** (*fused-multiply-add* – fusionar-multiplicar-sumar)
***from_float*(f)**	Devuelve una instancia de Decimal. Constructor alternativo que sólo acepta instancias de *float* o *int*.
d.*is_finite*()	Devuelve *True* si el operando es finito; en caso contrario devuelve *False*.
d.*is_infinite*()	Devuelve *True* si el operando es infinito; en caso contrario devuelve *False*.
d.*is_normal*(context=None)	Devuelve *True* si el operando es un número normal; en caso contrario devuelve *False*.
d.*is_signed*()	Devuelve *True* si el operando es negativo; en caso contrario devuelve *False*.
d.*is_zero*()	Devuelve *True* si el operando es un cero; en caso contrario devuelve *False*.
d.*ln*(context=None)	Devuelve el logaritmo natural (base e) del operando.
d.*log10*(context=None)	Devuelve el logaritmo en base 10 del operando.
d.*max*(other, context=None)	Devuelve el valor mayor.
d.*max_mag*(other, context=None)	Compara los valores numéricamente ignorando su signo y devuelve el máximo.
d.*min*(other, context=None)	Compara numéricamente dos valores y devuelve el mínimo.
d.*min_mag*(other, context=None)	Compara los valores numéricamente ignorando su signo y devuelve el mínimo.
d.*next_minus*(context=None)	Devuelve el mayor número representable menor que el operando en el contexto dado (o en el contexto del subproceso actual si no se da ningún contexto).

d.*next_plus*(context=None)	Devuelve el menor número representable mayor que el operando en el contexto dado (o en el contexto del subproceso actual si no se da ningún contexto).
d.*next_toward*(other, context=None)	Si los dos operandos son desiguales, devuelve el número más cercano al primer operando en la dirección del segundo operando. Si ambos operandos son numéricamente iguales, devuelve una copia del primer operando con el signo establecido para que sea el mismo que el signo del segundo operando.
d.*quantize*(exp, rounding=None, context=None)	Devuelve un valor igual al primer operando después de redondear que tiene el exponente del segundo operando.
d.*remainder_near*(other, context=None)	Devuelve el resto de la división entera. El valor de retorno es **self - n * other** donde n es el entero más cercano al valor exacto de **self/other**, y si dos enteros están igualmente cerca entonces se elige el par.
d.*sqrt*(context=None)	Devuelve la raíz cuadrada.
d.*to_eng_string*(context=None)	Convierte un número en una cadena, utilizando la notación científica. La notación científica/ingeniería tiene un exponente que es múltiplo de 3. Esto puede dejar hasta 3 dígitos a la izquierda del punto decimal y puede requerir la adición de uno o dos ceros finales.
d.*to_integral*(rounding=None, context=None)	Redondea a un entero. Idéntico al método *to_integral_value*(). Se ha mantenido por compatibilidad con versiones anteriores.
c.*to_integral_exact*(rounding=None, context=None)	Redondea a un entero. Los indicadores *Inexact* y *Rounded* están permitidos en esta operación. El modo de redondeo se toma del contexto.
d.*to_integral_value*(rounding=None, context=None)	Redondea a un entero. Es como *to_integral_exact*(), pero no se establece ningún indicador. El modo de redondeo se toma del contexto.

2.3.3 Contexto

Los decimales se asocian siempre con un contexto que establece los parámetros y reglas, seleccionables por el usuario, que rigen los resultados de las operaciones aritméticas.

El módulo *decimal* recién importado tiene un contexto predefinido que contiene valores por defecto para la precisión, el redondeo, los números mínimos y máximos permitidos, qué señales se tratan como excepciones y limitan el rango de los exponentes.

Por defecto, el contexto es global. Además, se puede establecer un contexto temporal que tendrá efecto localmente sin afectar al contexto global. Un contexto local es útil si se desea realizar una operación aritmética en un entorno temporal sin cambiar el entorno global.

La forma habitual de trabajo con decimales, además de importar el módulo, incluye crear un contexto de trabajo y, si es necesario, establecer nuevos valores para los diferentes atributos del contexto.

Es posible crear un contexto específico con la clase *Context*().

Clase	Descripción
ctx = decimal.*Context*(prec=None, rounding=None, Emin=None, Emax=None, capitals=None, clamp=None, flags=None, traps=None)	Crea un nuevo contexto. Si un campo no se especifica o es *None*, los valores por defecto se copian del *DefaultContext*. Si el campo **flags** no se especifica o es *None*, se borran todos los indicadores. **prec** es un número entero en el rango [1, MAX_PREC] que establece el número máximo de dígitos significativos que pueden resultar de una operación aritmética en el contexto. **rounding** establece la opción de redondeo según una de las constantes listadas en la tabla de Modos de redondeo. Los parámetros **traps** y **flags** enumeran las señales que se deben establecer. Por lo general, los nuevos contextos sólo deben establecer capturas y dejar los indicadores libres. Los parámetros **Emin** y **Emax** son enteros que especifican los límites exteriores permitidos para los exponentes. **Emin** debe estar en el rango [MIN_EMIN, 0], **Emax** en el rango [0, MAX_EMAX]. El parámetro **capitals** puede ser 0 o 1 (por defecto). Si se establece en 1, los exponentes se imprimen con una **E** mayúscula; en caso contrario, se utiliza una **e** minúscula. El parámetro **clamp** es 0 (por defecto) o 1. Si se establece en 1, el exponente **e** de una instancia de Decimal representable en este contexto se limita estrictamente al rango **Emin - prec + 1 <= e <= Emax - prec + 1**. Si es 0, se cumple una condición más débil: el exponente ajustado de la instancia Decimal es como máximo **Emax**. Si el valor de **clamp** es 1, se reducirá el exponente de un número normal grande y se añadirá el número correspondiente de ceros a su coeficiente, con el fin de ajustarlo a las restricciones del exponente; de este modo se conserva el valor del número, pero se pierde la información sobre los ceros finales significativos.

Python proporciona tres contextos ya preparados.

Contexto	Descripción
decimal.*BasicContext*	Los valores del contexto básico son: **Context(prec=9, rounding=ROUND_HALF_UP, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[], traps=[Clamped, InvalidOperation, DivisionByZero, Overflow, Underflow])** Debido a que muchas de las trampas están habilitadas, este contexto es útil para la depuración.
decimal.*ExtendedContext*	Los valores del contexto extendido son: **Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[], traps=[])** No se activan trampas, para que no se produzcan excepciones durante los cálculos. Debido a que las trampas están deshabilitadas, este contexto es útil para aplicaciones que prefieren tener un valor de resultado de *NaN* o Infinito en lugar de lanzar excepciones. Esto permite a una aplicación completar una ejecución en presencia de condiciones que de otro modo detendrían el programa.
decimal.*DefaultContext*	Los valores del contexto por defecto son: **Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[], traps=[InvalidOperation, DivisionByZero, Overflow])** El constructor *Context*() emplea este contexto como prototipo para nuevos contextos. Este contexto es muy útil en entornos multihilo. Cambiar uno de los campos antes de que se inicien los subprocesos tiene el efecto de establecer los valores por defecto de todo el sistema. No se recomienda cambiar los campos una vez iniciados los subprocesos, ya que requeriría la sincronización de los subprocesos para evitar condiciones de carrera. En entornos con un único subproceso, es preferible no utilizar este contexto. En su lugar, es mejor crear contextos explícitamente.

Un grupo de métodos facilita el manejo del contexto en curso.

Función	Descripción
decimal.*getcontext*()	Devuelve el contexto actual para el hilo activo. Si este hilo aún no tiene un contexto, devuelve un nuevo contexto y establece el contexto de ese hilo. Los nuevos contextos son copias de *DefaultContext*.
decimal.*setcontext*(ctx)	Establece el contexto actual para el hilo activo al contexto indicado por **ctx**.
decimal.*localcontext*(ctx=None,**kwargs)	Cambia temporalmente el contexto activo. Devuelve un gestor de contexto que establecerá el contexto actual para el hilo activo a una copia de **ctx** al entrar en la sentencia *with* y restaurará el contexto anterior al salir de la sentencia *with*. Si no se especifica ningún contexto, se utilizará una copia del contexto actual. El argumento **kwargs** se utiliza para establecer los atributos del nuevo contexto.