3. Std. module ‘math’ & ‘cmath’ – basic math functions

This module provides access to the **mathematical functions defined by the C standard**. These functions cannot be used with complex numbers; **use the functions of the same name from the ‘cmath‘ module if you require support for complex numbers.**

The ‘math‘ module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases follows Annex F of the C99 standard where appropriate.

```
import math
```

1 Number-theoretic and representation functions

Ceiling of x – the smallest integer greater than or equal to x:

```
math.ceil(x)
```

Floor of x – the largest integer less than or equal to x:

```
math.floor(x)
```

The integer value x truncated to an Integral (usually an integer):

```
math.trunc(x)
```

A float with the magnitude (absolute value) of x but the sign of y. On platforms that support signed zeros, copysign(1.0, -0.0) returns -1.0:

```
math.copysign(x, y)
```

Absolute value of x:

```
math.fabs(x)
abs()
```

Accurate floating point sum of values in the iterable. **Avoids loss of precision** by tracking multiple intermediate partial sums:

```
math.fsum(iterable)

sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
>>> 0.9999999999999999
math.fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
>>> 1.0
```

x factorial as an integer. Raises *ValueError* if x is not integral or is negative”

```
math.factorial(x)
```

Greatest common divisor of the integers a and b. If either a or b is nonzero, then the value of `gcd(a, b)` is the largest positive integer that divides both a and b:

```
math.gcd(a, b)
```

Return `True` if the values a and b are close to each other and `False` otherwise.

`rel_tol` is the relative tolerance – maximum allowed difference between a and b, relative to the larger absolute value of a or b. E.g., to set a tolerance of 5%, pass `rel_tol=0.05`. `rel_tol` must be greater than zero.

`abs_tol` is the minimum absolute tolerance – useful for comparisons near zero. `abs_tol` must be at least zero. If no errors occur, the result will be: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`

```
math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
```

`True` if x is neither an infinity nor a `NaN`, and `False` otherwise:

```
math.isfinite(x)
```

`True` if x is a positive or negative infinity, and `False` otherwise:

```
math.isinf(x)
```

`True` if x is a `NaN`, and `False` otherwise:

```
math.isnan(x)
```

Return the fractional and integer parts of x. Both results carry the sign of x and are floats:

```
math.modf(x)
```

IEEE 754-style remainder of x with respect to y. For finite x and finite nonzero y, this is the difference `x − n*y`, where n is the closest integer to the exact value of the quotient x / y:

```
math.remainder(x, y)
```

Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result.

For this reason, `fmod()` is generally preferred when working with floats, while Python's `x % y` is preferred when working with integers.

```
math.fmod(x, y)
```

2 Power and logarithmic functions

Return e raised to the power x:

```
math.exp(x)
```

Return e raised to the power x, minus 1. Here e is the base of natural logarithms. For small floats x, the subtraction in *exp(x) – 1* can result in a significant loss of precision; the `expm1()` function provides a way to compute this quantity to full precision:

```
math.expm1(x)
```

– with 1 argument: **natural logarithm** of x (to the base e).
– with 2 arguments: logarithm of x to the given base, calculated as *log(x)/log(base)*.

```
math.log(x[, base])
```

Return the **base-2 logarithm** of x. This is usually **more accurate than log(x, 2)**:

```
math.log2(x)
```

Return the **base-10 logarithm** of x. This is usually **more accurate than log(x, 10)**:

```
math.log10(x)
```

Return x raised to the power y.

In particular, `pow(1.0, y)` and `pow(x, 0.0)` always return `1.0`, even when x is a zero or a `NaN`. If both x and y are finite, x is negative, and y is not an integer then `pow(x, y)` is undefined, and raises *ValueError*.

Unlike the built-in operator, `math.pow()` converts both its arguments to type float.

```
math.pow(x, y)
```

Return the square root of x:

```
math.sqrt(x)
```

3 Trigonometric functions

Euclidean norm, `sqrt(x*x + y*y)`, i.e. the length of the vector from the origin to point `(x, y)`:

```
math.hypot(x, y)
```

cos of x radians:

```
math.cos(x)
```

sin of x radians:

```
math.sin(x)
```

tan of x radians:

```
math.tan(x)
```

acos of x, in radians:

```
math.acos(x)
```

asin of x, in radians:

```
math.asin(x)
```

atan of x, in radians:

```
math.atan(x)
```

Return `atan(y/x)`, in radians: $[-\pi, \pi]$:

```
math.atan2(y, x)
```

4 Angular conversions

Convert angle x **from radians to degrees**:

```
math.degrees(x)
```

Convert angle x **from degrees to radians**:

```
math.radians(x)
```

5 Special functions

The **error function at** $\boldsymbol{x}$. `erf()` can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

```
# Cumulative distribution function for the standard normal distribution:
def phi(x):
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

The **complementary error function at** $\boldsymbol{x}$. Defined as `1.0 - erf(x)`. Used for large x where a subtraction from one causes a loss of significance:

```
math.erfc(x)
```

The **Gamma function** at x:

```
math.gamma(x)
```

6 Constants

Constant $\pi = 3.141592\ldots$, to available precision:

```
math.pi
```

Constant $e = 2.718281\ldots$, to available precision:

```
math.e
```

A floating-point **positive infinity**. (For **negative infinity**, use −`math.inf`). Equivalent to the output of `float('inf')`:

```
math.inf
```

A floating-point “not a number” (`NaN`) value. Equivalent to the output of `float('nan')`:

```
math.nan
```

7 cmath — math functions for complex numbers

This module provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments.

Most of the fuctions have the same names, just additional part should refer to `cmath` instead of `math`.

```
import cmath
```

However:

```
z.real()
x.imag()
```

Representation of $\boldsymbol{x}$ **in polar coordinates**. Returns a pair `(r, phi)` where `r` is the modulus of x and `phi` is the phase (in rad) of x `.polar(x)` is equivalent to `(abs(x), cmath.phase(x))`:

```
cmath.polar(x)
```

Return the complex number $\boldsymbol{x}$ **with polar coordinates** `r` and `phi`.
Equivalent to `r * (math.cos(phi) + math.sin(phi)*1j)`:

```
cmath.rect(r, phi)
```

Return the **phase of** $\boldsymbol{x}$ **as a float**. Equivalent to `math.atan2(x.imag,x.real)`. The result is in the range $[-\pi, \pi]$:

```
cmath.phase(x)
```