CHAPTER

NINE

NUMERIC AND MATHEMATICAL MODULES

The modules described in this chapter provide numeric and math-related functions and data types. The *numbers* module defines an abstract hierarchy of numeric types. The *math* and *cmath* modules contain various mathematical functions for floating-point and complex numbers. The *decimal* module supports exact representations of decimal numbers, using arbitrary precision arithmetic.

The following modules are documented in this chapter:

9.1 numbers — Numeric abstract base classes

Source code: Lib/numbers.py

The *numbers* module (**PEP 3141**) defines a hierarchy of numeric *abstract base classes* which progressively define more operations. None of the types defined in this module can be instantiated.

class numbers.Number
The root of the numeric hierarchy. If you just want to check if an argument x is a number, without caring what kind, use `isinstance(x, Number)`.

9.1.1 The numeric tower

class numbers.Complex
Subclasses of this type describe complex numbers and include the operations that work on the built-in *complex* type. These are: conversions to *complex* and *bool*, *real*, *imag*, `+`, `-`, `*`, `/`, *abs()*, *conjugate()*, `==`, and `!=`. All except `-` and `!=` are abstract.

real
Abstract. Retrieves the real component of this number.

imag
Abstract. Retrieves the imaginary component of this number.

abstractmethod conjugate()
Abstract. Returns the complex conjugate. For example, `(1+3j).conjugate() == (1-3j)`.

class numbers.Real
To *Complex*, *Real* adds the operations that work on real numbers.

In short, those are: a conversion to *float*, *math.trunc()*, *round()*, *math.floor()*, *math.ceil()*, *divmod()*, `//`, `%`, `<`, `<=`, `>`, and `>=`.

Real also provides defaults for *complex()*, *real*, *imag*, and *conjugate()*.

class numbers.Rational
Subtypes *Real* and adds *numerator* and *denominator* properties, which should be in lowest terms. With these, it provides a default for *float()*.

numerator
Abstract.

denominator
Abstract.

class numbers.Integral
Subtypes *Rational* and adds a conversion to *int*. Provides defaults for *float()*, *numerator*, and *denominator*. Adds abstract methods for ** and bit-string operations: <<, >>, &, ^, |, ~.

9.1.2 Notes for type implementors

Implementors should be careful to make equal numbers equal and hash them to the same values. This may be subtle if there are two different extensions of the real numbers. For example, *fractions.Fraction* implements *hash()* as follows:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

Adding More Numeric ABCs

There are, of course, more possible ABCs for numbers, and this would be a poor hierarchy if it precluded the possibility of adding those. You can add MyFoo between *Complex* and *Real* with:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

Implementing the arithmetic operations

We want to implement the arithmetic operations so that mixed-mode operations either call an implementation whose author knew about the types of both arguments, or convert both to the nearest built in type and do the operation there. For subtypes of *Integral*, this means that __add__() and __radd__() should be defined as:

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
```

(continues on next page)

(continued from previous page)

```
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

There are 5 different cases for a mixed-type operation on subclasses of *Complex*. I'll refer to all of the above code that doesn't refer to `MyIntegral` and `OtherTypeIKnowAbout` as "boilerplate". `a` will be an instance of `A`, which is a subtype of *Complex* (`a : A <: Complex`), and `b : B <: Complex`. I'll consider `a + b`:

1. If `A` defines an `__add__()` which accepts `b`, all is well.
2. If `A` falls back to the boilerplate code, and it were to return a value from `__add__()`, we'd miss the possibility that `B` defines a more intelligent `__radd__()`, so the boilerplate should return *NotImplemented* from `__add__()`. (Or `A` may not implement `__add__()` at all.)
3. Then `B`'s `__radd__()` gets a chance. If it accepts `a`, all is well.
4. If it falls back to the boilerplate, there are no more possible methods to try, so this is where the default implementation should live.
5. If `B <: A`, Python tries `B.__radd__` before `A.__add__`. This is ok, because it was implemented with knowledge of `A`, so it can handle those instances before delegating to *Complex*.

If `A <: Complex` and `B <: Real` without sharing any other knowledge, then the appropriate shared operation is the one involving the built in *complex*, and both `__radd__()` s land there, so `a+b == b+a`.

Because most of the operations on any given type will be very similar, it can be useful to define a helper function which generates the forward and reverse instances of any given operator. For example, *fractions.Fraction* uses:

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__
```

(continues on next page)

(continued from previous page)

```
    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 `math` — Mathematical functions

This module is always available. It 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 distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

9.2.1 Number-theoretic and representation functions

`math.ceil(`*x*`)`
: Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to `x.__ceil__()`, which should return an *Integral* value.

`math.copysign(`*x*, *y*`)`
: Return 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.fabs(`*x*`)`
: Return the absolute value of *x*.

math.factorial(*x*)
Return *x* factorial. Raises *ValueError* if *x* is not integral or is negative.

math.floor(*x*)
Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to `x.__floor__()`, which should return an *Integral* value.

math.fmod(*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. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer *n* such that the result has the same sign as *x* and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of *y* instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented exactly as a float, and rounds to the surprising `1e100`. For this reason, function *fmod()* is generally preferred when working with floats, while Python's `x % y` is preferred when working with integers.

math.frexp(*x*)
Return the mantissa and exponent of *x* as the pair `(m, e)`. *m* is a float and *e* is an integer such that `x == m * 2**e` exactly. If *x* is zero, returns `(0.0, 0)`, otherwise `0.5 <= abs(m) < 1`. This is used to "pick apart" the internal representation of a float in a portable way.

math.fsum(*iterable*)
Return an accurate floating point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums:

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

The algorithm's accuracy depends on IEEE-754 arithmetic guarantees and the typical case where the rounding mode is half-even. On some non-Windows builds, the underlying C library uses extended precision addition and may occasionally double-round an intermediate sum causing it to be off in its least significant bit.

For further discussion and two alternative approaches, see the ASPN cookbook recipes for accurate floating point summation.

math.gcd(*a*, *b*)
Return the 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*. `gcd(0, 0)` returns `0`.

New in version 3.5.

math.isclose(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)
Return `True` if the values *a* and *b* are close to each other and `False` otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances.

rel_tol is the relative tolerance – it is the maximum allowed difference between *a* and *b*, relative to the larger absolute value of *a* or *b*. For example, to set a tolerance of 5%, pass `rel_tol=0.05`. The default tolerance is `1e-09`, which assures that the two values are the same within about 9 decimal digits. *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)`.

The IEEE 754 special values of `NaN`, `inf`, and `-inf` will be handled according to IEEE rules. Specifically, `NaN` is not considered close to any other value, including `NaN`. `inf` and `-inf` are only considered close to themselves.

New in version 3.5.

See also:

PEP 485 – A function for testing approximate equality

`math.isfinite(`*x*`)`
Return `True` if *x* is neither an infinity nor a NaN, and `False` otherwise. (Note that `0.0` *is* considered finite.)

New in version 3.2.

`math.isinf(`*x*`)`
Return `True` if *x* is a positive or negative infinity, and `False` otherwise.

`math.isnan(`*x*`)`
Return `True` if *x* is a NaN (not a number), and `False` otherwise.

`math.ldexp(`*x*, *i*`)`
Return `x * (2**i)`. This is essentially the inverse of function *frexp()*.

`math.modf(`*x*`)`
Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.

`math.remainder(`*x*, *y*`)`
Return the 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`. If `x / y` is exactly halfway between two consecutive integers, the nearest *even* integer is used for `n`. The remainder `r = remainder(x, y)` thus always satisfies `abs(r) <= 0.5 * abs(y)`.

Special cases follow IEEE 754: in particular, `remainder(x, math.inf)` is *x* for any finite *x*, and `remainder(x, 0)` and `remainder(math.inf, x)` raise *ValueError* for any non-NaN *x*. If the result of the remainder operation is zero, that zero will have the same sign as *x*.

On platforms using IEEE 754 binary floating-point, the result of this operation is always exactly representable: no rounding error is introduced.

New in version 3.7.

`math.trunc(`*x*`)`
Return the *Real* value *x* truncated to an *Integral* (usually an integer). Delegates to `x.__trunc__()`.

Note that *frexp()* and *modf()* have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an 'output parameter' (there is no such thing in Python).

For the *ceil()*, *floor()*, and *modf()* functions, note that *all* floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float *x* with `abs(x) >= 2**52` necessarily has no fractional bits.

9.2.2 Power and logarithmic functions

`math.exp(`*x*`)`
Return *e* raised to the power *x*, where *e* = 2.718281… is the base of natural logarithms. This is usually more accurate than `math.e ** x` or `pow(math.e, x)`.

`math.expm1(`*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:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

New in version 3.2.

math.log(*x*[, *base*])
With one argument, return the natural logarithm of *x* (to base *e*).

With two arguments, return the logarithm of *x* to the given *base*, calculated as `log(x)/log(base)`.

math.log1p(*x*)
Return the natural logarithm of *1+x* (base *e*). The result is calculated in a way which is accurate for *x* near zero.

math.log2(*x*)
Return the base-2 logarithm of *x*. This is usually more accurate than `log(x, 2)`.

New in version 3.3.

See also:

int.bit_length() returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

math.log10(*x*)
Return the base-10 logarithm of *x*. This is usually more accurate than `log(x, 10)`.

math.pow(*x*, *y*)
Return `x` raised to the power `y`. Exceptional cases follow Annex 'F' of the C99 standard as far as possible. In particular, `pow(1.0, x)` 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*. Use `**` or the built-in *pow()* function for computing exact integer powers.

math.sqrt(*x*)
Return the square root of *x*.

9.2.3 Trigonometric functions

math.acos(*x*)
Return the arc cosine of *x*, in radians.

math.asin(*x*)
Return the arc sine of *x*, in radians.

math.atan(*x*)
Return the arc tangent of *x*, in radians.

math.atan2(*y*, *x*)
Return `atan(y / x)`, in radians. The result is between `-pi` and `pi`. The vector in the plane from the origin to point `(x, y)` makes this angle with the positive X axis. The point of *atan2()* is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, `atan(1)` and `atan2(1, 1)` are both `pi/4`, but `atan2(-1, -1)` is `-3*pi/4`.

math.cos(*x*)
Return the cosine of *x* radians.

math.hypot(*x*, *y*)
Return the Euclidean norm, `sqrt(x*x + y*y)`. This is the length of the vector from the origin to point `(x, y)`.

math.sin(*x*)
Return the sine of *x* radians.

math.tan(*x*)
Return the tangent of *x* radians.

9.2.4 Angular conversion

math.degrees(*x*)
Convert angle *x* from radians to degrees.

math.radians(*x*)
Convert angle *x* from degrees to radians.

9.2.5 Hyperbolic functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

math.acosh(*x*)
Return the inverse hyperbolic cosine of *x*.

math.asinh(*x*)
Return the inverse hyperbolic sine of *x*.

math.atanh(*x*)
Return the inverse hyperbolic tangent of *x*.

math.cosh(*x*)
Return the hyperbolic cosine of *x*.

math.sinh(*x*)
Return the hyperbolic sine of *x*.

math.tanh(*x*)
Return the hyperbolic tangent of *x*.

9.2.6 Special functions

math.erf(*x*)
Return the error function at *x*.

The *erf()* function can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

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

New in version 3.2.

`math.erfc(`*x*`)`
Return the complementary error function at *x*. The complementary error function is defined as `1.0 - erf(x)`. It is used for large values of *x* where a subtraction from one would cause a loss of significance.

New in version 3.2.

`math.gamma(`*x*`)`
Return the Gamma function at *x*.

New in version 3.2.

`math.lgamma(`*x*`)`
Return the natural logarithm of the absolute value of the Gamma function at *x*.

New in version 3.2.

9.2.7 Constants

`math.pi`
The mathematical constant $\pi = 3.141592\ldots$, to available precision.

`math.e`
The mathematical constant $e = 2.718281\ldots$, to available precision.

`math.tau`
The mathematical constant $\tau = 6.283185\ldots$, to available precision. Tau is a circle constant equal to 2π, the ratio of a circle's circumference to its radius. To learn more about Tau, check out Vi Hart's video Pi is (still) Wrong, and start celebrating Tau day by eating twice as much pie!

New in version 3.6.

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

New in version 3.5.

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

New in version 3.5.

CPython implementation detail: 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. The current implementation will raise *ValueError* for invalid operations like `sqrt(-1.0)` or `log(0.0)` (where C99 Annex F recommends signaling invalid operation or divide-by-zero), and *OverflowError* for results that overflow (for example, `exp(1000.0)`). A NaN will not be returned from any of the functions above unless one or more of the input arguments was a NaN; in that case, most functions will return a NaN, but (again following C99 Annex F) there are some exceptions to this rule, for example `pow(float('nan'), 0.0)` or `hypot(float('nan'), float('inf'))`.

Note that Python makes no effort to distinguish signaling NaNs from quiet NaNs, and behavior for signaling NaNs remains unspecified. Typical behavior is to treat all NaNs as though they were quiet.

See also:

Module *cmath* Complex number versions of many of these functions.

9.3 cmath — Mathematical functions for complex numbers

This module is always available. It provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a `__complex__()` or a `__float__()` method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

Note: On platforms with hardware and system-level support for signed zeros, functions involving branch cuts are continuous on *both* sides of the branch cut: the sign of the zero distinguishes one side of the branch cut from the other. On platforms that do not support signed zeros the continuity is as specified below.

9.3.1 Conversions to and from polar coordinates

A Python complex number `z` is stored internally using *rectangular* or *Cartesian* coordinates. It is completely determined by its *real part* `z.real` and its *imaginary part* `z.imag`. In other words:

```
z == z.real + z.imag*1j
```

Polar coordinates give an alternative way to represent a complex number. In polar coordinates, a complex number z is defined by the modulus r and the phase angle *phi*. The modulus r is the distance from z to the origin, while the phase *phi* is the counterclockwise angle, measured in radians, from the positive x-axis to the line segment that joins the origin to z.

The following functions can be used to convert from the native rectangular coordinates to polar coordinates and back.

`cmath.phase(`x`)`
: Return the phase of x (also known as the *argument* of x), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range [$-\pi$, π], and the branch cut for this operation lies along the negative real axis, continuous from above. On systems with support for signed zeros (which includes most systems in current use), this means that the sign of the result is the same as the sign of `x.imag`, even when `x.imag` is zero:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

Note: The modulus (absolute value) of a complex number x can be computed using the built-in *abs()* function. There is no separate *cmath* module function for this operation.

`cmath.polar(`x`)`
: Return the representation of x in polar coordinates. Returns a pair `(r, phi)` where r is the modulus of x and phi is the phase of x. `polar(x)` is equivalent to `(abs(x), phase(x))`.

`cmath.rect(`r, *phi*`)`
: Return the complex number x with polar coordinates r and *phi*. Equivalent to `r * (math.cos(phi) + math.sin(phi)*1j)`.

9.3.2 Power and logarithmic functions

`cmath.exp(`x`)`
: Return e raised to the power x, where e is the base of natural logarithms.

cmath.log(*x*[, *base*])
Returns the logarithm of *x* to the given *base*. If the *base* is not specified, returns the natural logarithm of *x*. There is one branch cut, from 0 along the negative real axis to -∞, continuous from above.

cmath.log10(*x*)
Return the base-10 logarithm of *x*. This has the same branch cut as *log()*.

cmath.sqrt(*x*)
Return the square root of *x*. This has the same branch cut as *log()*.

9.3.3 Trigonometric functions

cmath.acos(*x*)
Return the arc cosine of *x*. There are two branch cuts: One extends right from 1 along the real axis to ∞, continuous from below. The other extends left from -1 along the real axis to -∞, continuous from above.

cmath.asin(*x*)
Return the arc sine of *x*. This has the same branch cuts as *acos()*.

cmath.atan(*x*)
Return the arc tangent of *x*. There are two branch cuts: One extends from `1j` along the imaginary axis to ∞`j`, continuous from the right. The other extends from `-1j` along the imaginary axis to -∞`j`, continuous from the left.

cmath.cos(*x*)
Return the cosine of *x*.

cmath.sin(*x*)
Return the sine of *x*.

cmath.tan(*x*)
Return the tangent of *x*.

9.3.4 Hyperbolic functions

cmath.acosh(*x*)
Return the inverse hyperbolic cosine of *x*. There is one branch cut, extending left from 1 along the real axis to -∞, continuous from above.

cmath.asinh(*x*)
Return the inverse hyperbolic sine of *x*. There are two branch cuts: One extends from `1j` along the imaginary axis to ∞`j`, continuous from the right. The other extends from `-1j` along the imaginary axis to -∞`j`, continuous from the left.

cmath.atanh(*x*)
Return the inverse hyperbolic tangent of *x*. There are two branch cuts: One extends from `1` along the real axis to ∞, continuous from below. The other extends from `-1` along the real axis to -∞, continuous from above.

cmath.cosh(*x*)
Return the hyperbolic cosine of *x*.

cmath.sinh(*x*)
Return the hyperbolic sine of *x*.

cmath.tanh(*x*)
Return the hyperbolic tangent of *x*.

9.3.5 Classification functions

`cmath.isfinite(`*x*`)`
Return `True` if both the real and imaginary parts of *x* are finite, and `False` otherwise.

New in version 3.2.

`cmath.isinf(`*x*`)`
Return `True` if either the real or the imaginary part of *x* is an infinity, and `False` otherwise.

`cmath.isnan(`*x*`)`
Return `True` if either the real or the imaginary part of *x* is a NaN, and `False` otherwise.

`cmath.isclose(`*a*, *b*, *, *rel_tol=1e-09*, *abs_tol=0.0*`)`
Return `True` if the values *a* and *b* are close to each other and `False` otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances.

rel_tol is the relative tolerance – it is the maximum allowed difference between *a* and *b*, relative to the larger absolute value of *a* or *b*. For example, to set a tolerance of 5%, pass `rel_tol=0.05`. The default tolerance is `1e-09`, which assures that the two values are the same within about 9 decimal digits. *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)`.

The IEEE 754 special values of `NaN`, `inf`, and `-inf` will be handled according to IEEE rules. Specifically, `NaN` is not considered close to any other value, including `NaN`. `inf` and `-inf` are only considered close to themselves.

New in version 3.5.

See also:

PEP 485 – A function for testing approximate equality

9.3.6 Constants

`cmath.pi`
The mathematical constant π, as a float.

`cmath.e`
The mathematical constant e, as a float.

`cmath.tau`
The mathematical constant τ, as a float.

New in version 3.6.

`cmath.inf`
Floating-point positive infinity. Equivalent to `float('inf')`.

New in version 3.6.

`cmath.infj`
Complex number with zero real part and positive infinity imaginary part. Equivalent to `complex(0.0, float('inf'))`.

New in version 3.6.

`cmath.nan`
A floating-point "not a number" (NaN) value. Equivalent to `float('nan')`.

New in version 3.6.

`cmath.nanj`
Complex number with zero real part and NaN imaginary part. Equivalent to `complex(0.0, float('nan'))`.

New in version 3.6.

Note that the selection of functions is similar, but not identical, to that in module *math*. The reason for having two modules is that some users aren't interested in complex numbers, and perhaps don't even know what they are. They would rather have `math.sqrt(-1)` raise an exception than return a complex number. Also note that the functions defined in *cmath* always return a complex number, even if the answer can be expressed as a real number (in which case the complex number has an imaginary part of zero).

A note on branch cuts: They are curves along which the given function fails to be continuous. They are a necessary feature of many complex functions. It is assumed that if you need to compute with complex functions, you will understand about branch cuts. Consult almost any (not too elementary) book on complex variables for enlightenment. For information of the proper choice of branch cuts for numerical purposes, a good reference should be the following:

See also:

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165–211.

9.4 `decimal` — Decimal fixed point and floating point arithmetic

Source code: Lib/decimal.py

The *decimal* module provides support for fast correctly-rounded decimal floating point arithmetic. It offers several advantages over the *float* datatype:

- Decimal "is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school." – excerpt from the decimal arithmetic specification.
- Decimal numbers can be represented exactly. In contrast, numbers like `1.1` and `2.2` do not have exact representations in binary floating point. End users typically would not expect `1.1 + 2.2` to display as `3.3000000000000003` as it does with binary floating point.
- The exactness carries over into arithmetic. In decimal floating point, `0.1 + 0.1 + 0.1 - 0.3` is exactly equal to zero. In binary floating point, the result is `5.5511151231257827e-017`. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.
- The decimal module incorporates a notion of significant places so that `1.30 + 1.20` is `2.50`. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the "schoolbook" approach uses all the figures in the multiplicands. For instance, `1.3 * 1.2` gives `1.56` while `1.30 * 1.20` gives `1.5600`.
- Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.
- The decimal module was designed to support "without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic." – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as `Infinity`, `-Infinity`, and `NaN`. The standard also differentiates `-0` from `+0`.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include *ROUND_CEILING*, *ROUND_DOWN*, *ROUND_FLOOR*, *ROUND_HALF_DOWN*, *ROUND_HALF_EVEN*, *ROUND_HALF_UP*, *ROUND_UP*, and *ROUND_05UP*.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: *Clamped*, *InvalidOperation*, *DivisionByZero*, *Inexact*, *Rounded*, *Subnormal*, *Overflow*, *Underflow* and *FloatOperation*.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

See also:

- IBM's General Decimal Arithmetic Specification, The General Decimal Arithmetic Specification.

9.4.1 Quick-start Tutorial

The usual start to using decimals is importing the module, viewing the current context with *getcontext()* and, if necessary, setting new values for precision, rounding, or enabled traps:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as `NaN` which stands for "Not a number", positive and negative `Infinity`, and `-0`:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

If the *FloatOperation* signal is trapped, accidental mixing of decimals and floats in constructors or ordering comparisons raises an exception:

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

New in version 3.3.

The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

If the internal limits of the C version are exceeded, constructing a decimal raises *InvalidOperation*:

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

Changed in version 3.3.

Decimals interact well with much of the rest of Python. Here is a small decimal floating point flying circus:

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

And some mathematical functions are also available to Decimal:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

The `quantize()` method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

As shown above, the *getcontext()* function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.

For more advanced work, it may be useful to create alternate contexts using the Context() constructor. To make an alternate active, use the *setcontext()* function.

In accordance with the standard, the *decimal* module provides two ready to use standard contexts, *BasicContext* and *ExtendedContext*. The former is especially useful for debugging because many of the traps are enabled:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the `clear_flags()` method.

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

The *flags* entry shows that the rational approximation to `Pi` was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the `traps` field of a context:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
```

(continues on next page)

(continued from previous page)

```
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to *Decimal* with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.

9.4.2 Decimal objects

class decimal.**Decimal**(*value="0"*, *context=None*)

Construct a new *Decimal* object based from *value*.

value can be an integer, string, tuple, *float*, or another *Decimal* object. If no *value* is given, returns `Decimal('0')`. If *value* is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters, as well as underscores throughout, are removed:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

Other Unicode decimal digits are also permitted where `digit` appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits `'\uff10'` through `'\uff19'`.

If *value* is a *tuple*, it should have three components, a sign (`0` for positive or `1` for negative), a *tuple* of digits, and an integer exponent. For example, `Decimal((0, (1, 4, 1, 4), -3))` returns `Decimal('1.414')`.

If *value* is a *float*, the binary floating point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, `Decimal(float('1.1'))` converts to `Decimal('1.100000000000000088817841970012523233890533447265625')`.

The *context* precision does not affect how many digits are stored. That is determined exclusively by the number of digits in *value*. For example, `Decimal('3.00000')` records all five zeros even if the context precision is only three.

The purpose of the *context* argument is determining what to do if *value* is a malformed string. If the context traps *InvalidOperation*, an exception is raised; otherwise, the constructor returns a new Decimal with the value of `NaN`.

Once constructed, *Decimal* objects are immutable.

Changed in version 3.2: The argument to the constructor is now permitted to be a *float* instance.

Changed in version 3.3: *float* arguments raise an exception if the *FloatOperation* trap is set. By default the trap is off.

Changed in version 3.6: Underscores are allowed for grouping, as with integral and floating-point literals in code.

Decimal floating point objects share many properties with the other built-in numeric types such as *float* and *int*. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as *float* or *int*).

There are some small differences between arithmetic on Decimal objects and arithmetic on integers and floats. When the remainder operator `%` is applied to Decimal objects, the sign of the result is the sign of the *dividend* rather than the sign of the divisor:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

The integer division operator `//` behaves analogously, returning the integer part of the true quotient (truncating towards zero) rather than its floor, so as to preserve the usual identity `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

The `%` and `//` operators implement the `remainder` and `divide-integer` operations (respectively) as described in the specification.

Decimal objects cannot generally be combined with floats or instances of *fractions.Fraction* in arithmetic operations: an attempt to add a *Decimal* to a *float*, for example, will raise a *TypeError*. However, it is possible to use Python's comparison operators to compare a *Decimal* instance `x` with another number `y`. This avoids confusing results when doing equality comparisons between numbers of different types.

Changed in version 3.2: Mixed-type comparisons between *Decimal* instances and other numeric types are now fully supported.

In addition to the standard numeric properties, decimal floating point objects also have a number of specialized methods:

adjusted()
: Return the adjusted exponent after shifting out the coefficient's rightmost digits until only the lead digit remains: `Decimal('321e+5').adjusted()` returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

as_integer_ratio()
: Return a pair `(n, d)` of integers that represent the given *Decimal* instance as a fraction, in lowest terms and with a positive denominator:

  ```
  >>> Decimal('-3.14').as_integer_ratio()
  (-157, 50)
  ```

  The conversion is exact. Raise OverflowError on infinities and ValueError on NaNs.

New in version 3.6.

as_tuple()
: Return a *named tuple* representation of the number: `DecimalTuple(sign, digits, exponent)`.

canonical()
: Return the canonical encoding of the argument. Currently, the encoding of a *Decimal* instance is always canonical, so this operation returns its argument unchanged.

compare(*other*, *context=None*)
Compare the values of two Decimal instances. *compare()* returns a Decimal instance, and if either operand is a NaN then the result is a NaN:

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)
This operation is identical to the *compare()* method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

compare_total(*other*, *context=None*)
Compare two operands using their abstract representation rather than their numerical value. Similar to the *compare()* method, but the result gives a total ordering on *Decimal* instances. Two *Decimal* instances with the same numeric value but different representations compare unequal in this ordering:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is `Decimal('0')` if both operands have the same representation, `Decimal('-1')` if the first operand is lower in the total order than the second, and `Decimal('1')` if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

compare_total_mag(*other*, *context=None*)
Compare two operands using their abstract representation rather than their value as in *compare_total()*, but ignoring the sign of each operand. `x.compare_total_mag(y)` is equivalent to `x.copy_abs().compare_total(y.copy_abs())`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

conjugate()
Just returns self, this method is only to comply with the Decimal Specification.

copy_abs()
Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

copy_negate()
Return the negation of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

copy_sign(*other*, *context=None*)
Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

exp(*context=None*)
Return the value of the (natural) exponential function `e**x` at the given number. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

from_float(*f*)
Classmethod that converts a float to a decimal number, exactly.

Note *Decimal.from_float(0.1)* is not the same as *Decimal('0.1')*. Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is *0x1.999999999999ap-4*. That equivalent value in decimal is *0.1000000000000000055511151231257827021181583404541015625*.

Note: From Python 3.2 onwards, a *Decimal* instance can also be constructed directly from a *float*.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

New in version 3.1.

fma(*other*, *third*, *context=None*)
Fused multiply-add. Return self*other+third with no rounding of the intermediate product self*other.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()
Return *True* if the argument is canonical and *False* otherwise. Currently, a *Decimal* instance is always canonical, so this operation always returns *True*.

is_finite()
Return *True* if the argument is a finite number, and *False* if the argument is an infinity or a NaN.

is_infinite()
Return *True* if the argument is either positive or negative infinity and *False* otherwise.

is_nan()
Return *True* if the argument is a (quiet or signaling) NaN and *False* otherwise.

is_normal(*context=None*)
Return *True* if the argument is a *normal* finite number. Return *False* if the argument is zero, subnormal, infinite or a NaN.

is_qnan()
Return *True* if the argument is a quiet NaN, and *False* otherwise.

is_signed()
Return *True* if the argument has a negative sign and *False* otherwise. Note that zeros and NaNs can both carry signs.

is_snan()
Return *True* if the argument is a signaling NaN and *False* otherwise.

is_subnormal(*context=None*)
Return *True* if the argument is subnormal, and *False* otherwise.

is_zero()
Return *True* if the argument is a (positive or negative) zero and *False* otherwise.

ln(*context=None*)
Return the natural (base e) logarithm of the operand. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

log10(*context=None*)
Return the base ten logarithm of the operand. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

logb(*context=None*)
For a nonzero number, return the adjusted exponent of its operand as a *Decimal* instance. If the operand is a zero then `Decimal('-Infinity')` is returned and the *DivisionByZero* flag is raised. If the operand is an infinity then `Decimal('Infinity')` is returned.

logical_and(*other*, *context=None*)
logical_and() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise `and` of the two operands.

logical_invert(*context=None*)
logical_invert() is a logical operation. The result is the digit-wise inversion of the operand.

logical_or(*other*, *context=None*)
logical_or() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise `or` of the two operands.

logical_xor(*other*, *context=None*)
logical_xor() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise exclusive or of the two operands.

max(*other*, *context=None*)
Like `max(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

max_mag(*other*, *context=None*)
Similar to the *max()* method, but the comparison is done using the absolute values of the operands.

min(*other*, *context=None*)
Like `min(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

min_mag(*other*, *context=None*)
Similar to the *min()* method, but the comparison is done using the absolute values of the operands.

next_minus(*context=None*)
Return the largest number representable in the given context (or in the current thread's context if no context is given) that is smaller than the given operand.

next_plus(*context=None*)
Return the smallest number representable in the given context (or in the current thread's context if no context is given) that is larger than the given operand.

next_toward(*other*, *context=None*)
If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

normalize(*context=None*)
Normalize the number by stripping the rightmost trailing zeros and converting any result equal to `Decimal('0')` to `Decimal('0e0')`. Used for producing canonical values for attributes of an equivalence class. For example, `Decimal('32.100')` and `Decimal('0.321000e+2')` both normalize to the equivalent value `Decimal('32.1')`.

number_class(*context=None*)
Return a string describing the *class* of the operand. The returned value is one of the following ten strings.

- `"-Infinity"`, indicating that the operand is negative infinity.
- `"-Normal"`, indicating that the operand is a negative normal number.
- `"-Subnormal"`, indicating that the operand is negative and subnormal.
- `"-Zero"`, indicating that the operand is a negative zero.
- `"+Zero"`, indicating that the operand is a positive zero.
- `"+Subnormal"`, indicating that the operand is positive and subnormal.
- `"+Normal"`, indicating that the operand is a positive normal number.
- `"+Infinity"`, indicating that the operand is positive infinity.
- `"NaN"`, indicating that the operand is a quiet NaN (Not a Number).
- `"sNaN"`, indicating that the operand is a signaling NaN.

quantize(*exp*, *rounding=None*, *context=None*)
Return a value equal to the first operand after rounding and having the exponent of the second operand.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an *InvalidOperation* is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the `rounding` argument if given, else by the given `context` argument; if neither argument is given the rounding mode of the current thread's context is used.

An error is returned whenever the resulting exponent is greater than `Emax` or less than `Etiny`.

radix()
Return `Decimal(10)`, the radix (base) in which the *Decimal* class does all its arithmetic. Included for compatibility with the specification.

remainder_near(*other*, *context=None*)
Return the remainder from dividing *self* by *other*. This differs from `self % other` in that the sign of the remainder is chosen so as to minimize its absolute value. More precisely, the return value is `self - n * other` where `n` is the integer nearest to the exact value of `self / other`, and if two integers are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of *self*.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)
Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

same_quantum(*other*, *context=None*)
Test whether self and other have the same exponent or whether both are `NaN`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

scaleb(*other*, *context=None*)
Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by `10**other`. The second operand must be an integer.

shift(*other*, *context=None*)
Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

sqrt(*context=None*)
Return the square root of the argument to full precision.

to_eng_string(*context=None*)
Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

For example, this converts `Decimal('123E+1')` to `Decimal('1.23E+3')`.

to_integral(*rounding=None*, *context=None*)
Identical to the *to_integral_value()* method. The `to_integral` name has been kept for compatibility with older versions.

to_integral_exact(*rounding=None*, *context=None*)
Round to the nearest integer, signaling *Inexact* or *Rounded* as appropriate if rounding occurs. The rounding mode is determined by the `rounding` parameter if given, else by the given `context`. If neither parameter is given then the rounding mode of the current context is used.

to_integral_value(*rounding=None*, *context=None*)
Round to the nearest integer without signaling *Inexact* or *Rounded*. If given, applies *rounding*; otherwise, uses the rounding method in either the supplied *context* or the current context.

Logical operands

The `logical_and()`, `logical_invert()`, `logical_or()`, and `logical_xor()` methods expect their arguments to be *logical operands*. A *logical operand* is a *Decimal* instance whose exponent and sign are both zero, and whose digits are all either `0` or `1`.

9.4.3 Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the *getcontext()* and *setcontext()* functions:

decimal.getcontext()
Return the current context for the active thread.

decimal.setcontext(*c*)
Set the current context for the active thread to *c*.

You can also use the `with` statement and the *localcontext()* function to temporarily change the active context.

decimal.localcontext(*ctx=None*)
Return a context manager that will set the current context for the active thread to a copy of *ctx* on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used.

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

New contexts can also be created using the *Context* constructor described below. In addition, the module provides three pre-made contexts:

class decimal.BasicContext
This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to *ROUND_HALF_UP*. All flags are cleared. All traps are enabled (treated as exceptions) except *Inexact*, *Rounded*, and *Subnormal*.

Because many of the traps are enabled, this context is useful for debugging.

class decimal.ExtendedContext

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to *ROUND_HALF_EVEN*. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the traps are disabled, this context is useful for applications that prefer to have result value of `NaN` or `Infinity` instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

class decimal.DefaultContext

This context is used by the *Context* constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts created by the *Context* constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are `prec`=28, `rounding`=*ROUND_HALF_EVEN*, and enabled traps for *Overflow*, *InvalidOperation*, and *DivisionByZero*.

In addition to the three supplied contexts, new contexts can be created with the *Context* constructor.

class decimal.Context(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)

Creates a new context. If a field is not specified or is *None*, the default values are copied from the *DefaultContext*. If the *flags* field is not specified or is *None*, all flags are cleared.

prec is an integer in the range [`1`, *MAX_PREC*] that sets the precision for arithmetic operations in the context.

The *rounding* option is one of the constants listed in the section *Rounding Modes*.

The *traps* and *flags* fields list any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

The *Emin* and *Emax* fields are integers specifying the outer limits allowable for exponents. *Emin* must be in the range [*MIN_EMIN*, `0`], *Emax* in the range [`0`, *MAX_EMAX*].

The *capitals* field is either `0` or `1` (the default). If set to `1`, exponents are printed with a capital `E`; otherwise, a lowercase `e` is used: `Decimal('6.02e+23')`.

The *clamp* field is either `0` (the default) or `1`. If set to `1`, the exponent `e` of a *Decimal* instance representable in this context is strictly limited to the range `Emin - prec + 1 <= e <= Emax - prec + 1`. If *clamp* is `0` then a weaker condition holds: the adjusted exponent of the *Decimal* instance is at most `Emax`. When *clamp* is `1`, a large normal number will, where possible, have its exponent reduced and a corresponding number of zeros added to its coefficient, in order to fit the exponent constraints; this preserves the value of the number but loses information about significant trailing zeros. For example:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

A *clamp* value of `1` allows compatibility with the fixed-width decimal interchange formats specified in IEEE 754.

The *Context* class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the *Decimal* methods described above (with the exception of the `adjusted()` and `as_tuple()` methods) there is a corresponding *Context* method. For example, for a *Context* instance `C` and *Decimal* instance `x`, `C.exp(x)` is equivalent to

`x.exp(context=C)`. Each *Context* method accepts a Python integer (an instance of *int*) anywhere that a Decimal instance is accepted.

clear_flags()
Resets all of the flags to 0.

clear_traps()
Resets all of the traps to 0.

New in version 3.3.

copy()
Return a duplicate of the context.

copy_decimal(*num*)
Return a copy of the Decimal instance num.

create_decimal(*num*)
Creates a new Decimal instance from *num* but using *self* as context. Unlike the *Decimal* constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace or underscores are permitted.

create_decimal_from_float(*f*)
Creates a new Decimal instance from a float *f* but rounding using *self* as the context. Unlike the *Decimal.from_float()* class method, the context precision, rounding method, flags, and traps are applied to the conversion.

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

New in version 3.1.

Etiny()
Returns a value equal to `Emin - prec + 1` which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to *Etiny*.

Etop()
Returns a value equal to `Emax - prec + 1`.

The usual approach to working with decimals is to create *Decimal* instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach

is to use context methods for calculating within a specific context. The methods are similar to those for the *Decimal* class and are only briefly recounted here.

abs(*x*)
Returns the absolute value of *x*.

add(*x*, *y*)
Return the sum of *x* and *y*.

canonical(*x*)
Returns the same Decimal object *x*.

compare(*x*, *y*)
Compares *x* and *y* numerically.

compare_signal(*x*, *y*)
Compares the values of the two operands numerically.

compare_total(*x*, *y*)
Compares two operands using their abstract representation.

compare_total_mag(*x*, *y*)
Compares two operands using their abstract representation, ignoring sign.

copy_abs(*x*)
Returns a copy of *x* with the sign set to 0.

copy_negate(*x*)
Returns a copy of *x* with the sign inverted.

copy_sign(*x*, *y*)
Copies the sign from *y* to *x*.

divide(*x*, *y*)
Return *x* divided by *y*.

divide_int(*x*, *y*)
Return *x* divided by *y*, truncated to an integer.

divmod(*x*, *y*)
Divides two numbers and returns the integer part of the result.

exp(*x*)
Returns *e* ** *x*.

fma(*x*, *y*, *z*)
Returns *x* multiplied by *y*, plus *z*.

is_canonical(*x*)
Returns `True` if *x* is canonical; otherwise returns `False`.

is_finite(*x*)
Returns `True` if *x* is finite; otherwise returns `False`.

is_infinite(*x*)
Returns `True` if *x* is infinite; otherwise returns `False`.

is_nan(*x*)
Returns `True` if *x* is a qNaN or sNaN; otherwise returns `False`.

is_normal(*x*)
Returns `True` if *x* is a normal number; otherwise returns `False`.

is_qnan(*x*)
Returns `True` if *x* is a quiet NaN; otherwise returns `False`.

`is_signed`(*x*)
Returns `True` if *x* is negative; otherwise returns `False`.

`is_snan`(*x*)
Returns `True` if *x* is a signaling NaN; otherwise returns `False`.

`is_subnormal`(*x*)
Returns `True` if *x* is subnormal; otherwise returns `False`.

`is_zero`(*x*)
Returns `True` if *x* is a zero; otherwise returns `False`.

`ln`(*x*)
Returns the natural (base e) logarithm of *x*.

`log10`(*x*)
Returns the base 10 logarithm of *x*.

`logb`(*x*)
Returns the exponent of the magnitude of the operand's MSD.

`logical_and`(*x*, *y*)
Applies the logical operation *and* between each operand's digits.

`logical_invert`(*x*)
Invert all the digits in *x*.

`logical_or`(*x*, *y*)
Applies the logical operation *or* between each operand's digits.

`logical_xor`(*x*, *y*)
Applies the logical operation *xor* between each operand's digits.

`max`(*x*, *y*)
Compares two values numerically and returns the maximum.

`max_mag`(*x*, *y*)
Compares the values numerically with their sign ignored.

`min`(*x*, *y*)
Compares two values numerically and returns the minimum.

`min_mag`(*x*, *y*)
Compares the values numerically with their sign ignored.

`minus`(*x*)
Minus corresponds to the unary prefix minus operator in Python.

`multiply`(*x*, *y*)
Return the product of *x* and *y*.

`next_minus`(*x*)
Returns the largest representable number smaller than *x*.

`next_plus`(*x*)
Returns the smallest representable number larger than *x*.

`next_toward`(*x*, *y*)
Returns the number closest to *x*, in direction towards *y*.

`normalize`(*x*)
Reduces *x* to its simplest form.

`number_class`(*x*)
Returns an indication of the class of *x*.

plus(*x*)
Plus corresponds to the unary prefix plus operator in Python. This operation applies the context precision and rounding, so it is *not* an identity operation.

power(*x*, *y*, *modulo=None*)
Return `x` to the power of `y`, reduced modulo `modulo` if given.

With two arguments, compute `x**y`. If `x` is negative then `y` must be integral. The result will be inexact unless `y` is integral and the result is finite and can be expressed exactly in 'precision' digits. The rounding mode of the context is used. Results are always correctly-rounded in the Python version.

Changed in version 3.3: The C module computes *power()* in terms of the correctly-rounded *exp()* and *ln()* functions. The result is well-defined but only "almost always correctly-rounded".

With three arguments, compute `(x**y) % modulo`. For the three argument form, the following restrictions on the arguments hold:

- all three arguments must be integral
- `y` must be nonnegative
- at least one of `x` or `y` must be nonzero
- `modulo` must be nonzero and have at most 'precision' digits

The value resulting from `Context.power(x, y, modulo)` is equal to the value that would be obtained by computing `(x**y) % modulo` with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of `x`, `y` and `modulo`. The result is always exact.

quantize(*x*, *y*)
Returns a value equal to *x* (rounded), having the exponent of *y*.

radix()
Just returns 10, as this is Decimal, :)

remainder(*x*, *y*)
Returns the remainder from integer division.

The sign of the result, if non-zero, is the same as that of the original dividend.

remainder_near(*x*, *y*)
Returns `x - y * n`, where *n* is the integer nearest the exact value of `x / y` (if the result is 0 then its sign will be the sign of *x*).

rotate(*x*, *y*)
Returns a rotated copy of *x*, *y* times.

same_quantum(*x*, *y*)
Returns `True` if the two operands have the same exponent.

scaleb(*x*, *y*)
Returns the first operand after adding the second value its exp.

shift(*x*, *y*)
Returns a shifted copy of *x*, *y* times.

sqrt(*x*)
Square root of a non-negative number to context precision.

subtract(*x*, *y*)
Return the difference between *x* and *y*.

to_eng_string(*x*)
Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

`to_integral_exact`(*x*)
Rounds to an integer.

`to_sci_string`(*x*)
Converts a number to a string using scientific notation.

9.4.4 Constants

The constants in this section are only relevant for the C module. They are also included in the pure Python version for compatibility.

	32-bit	64-bit
`decimal.MAX_PREC`	425000000	999999999999999999
`decimal.MAX_EMAX`	425000000	999999999999999999
`decimal.MIN_EMIN`	-425000000	-999999999999999999
`decimal.MIN_ETINY`	-849999999	-1999999999999999997

`decimal.HAVE_THREADS`
The default value is `True`. If Python is compiled without threads, the C version automatically disables the expensive thread local context machinery. In this case, the value is `False`.

9.4.5 Rounding modes

`decimal.ROUND_CEILING`
Round towards `Infinity`.

`decimal.ROUND_DOWN`
Round towards zero.

`decimal.ROUND_FLOOR`
Round towards `-Infinity`.

`decimal.ROUND_HALF_DOWN`
Round to nearest with ties going towards zero.

`decimal.ROUND_HALF_EVEN`
Round to nearest with ties going to nearest even integer.

`decimal.ROUND_HALF_UP`
Round to nearest with ties going away from zero.

`decimal.ROUND_UP`
Round away from zero.

`decimal.ROUND_05UP`
Round away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise round towards zero.

9.4.6 Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the *DivisionByZero* trap is set, then a *DivisionByZero* exception is raised upon encountering the condition.

class decimal.Clamped
Altered an exponent to fit representation constraints.

Typically, clamping occurs when an exponent falls outside the context's `Emin` and `Emax` limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class decimal.DecimalException
Base class for other signals and a subclass of *ArithmeticError*.

class decimal.DivisionByZero
Signals the division of a non-infinite number by zero.

Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns `Infinity` or `-Infinity` with the sign determined by the inputs to the calculation.

class decimal.Inexact
Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

class decimal.InvalidOperation
An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns `NaN`. Possible causes include:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.Overflow
Numerical overflow.

Indicates the exponent is larger than `Emax` after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to `Infinity`. In either case, *Inexact* and *Rounded* are also signaled.

class decimal.Rounded
Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding `5.00` to `5.0`). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class decimal.Subnormal
Exponent was lower than `Emin` prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

class decimal.Underflow
Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. *Inexact* and *Subnormal* are also signaled.

class decimal.FloatOperation
Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the *Decimal* constructor, *create_decimal()* and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting *FloatOperation* in the context flags. Explicit conversions with *from_float()* or *create_decimal_from_float()* do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise *FloatOperation*.

The following table summarizes the hierarchy of signals:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 Floating Point Notes

Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent `0.1` exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8
```

(continues on next page)

(continued from previous page)

```
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

The *decimal* module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

Special values

The number system for the *decimal* module provides special values including `NaN`, `sNaN`, `-Infinity`, `Infinity`, and two zeros, `+0` and `-0`.

Infinities can be constructed directly with: `Decimal('Infinity')`. Also, they can arise from dividing by zero when the *DivisionByZero* signal is not trapped. Likewise, when the *Overflow* signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return `NaN`, or if the *InvalidOperation* signal is trapped, raise an exception. For example, `0/0` returns `NaN` which means "not a number". This variety of `NaN` is quiet and, once created, will flow through other computations always resulting in another `NaN`. This behavior can be useful for a series of computations that occasionally have missing inputs — it allows the calculation to proceed while flagging specific results as invalid.

A variant is `sNaN` which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python's comparison operators can be a little surprising where a `NaN` is involved. A test for equality where one of the operands is a quiet or signaling `NaN` always returns *False* (even when doing `Decimal('NaN')==Decimal('NaN')`), while a test for inequality always returns *True*. An attempt to compare two Decimals using any of the `<`, `<=`, `>` or `>=` operators will raise the *InvalidOperation* signal if either operand is a `NaN`, and return *False* if this signal is not trapped. Note that the General Decimal Arithmetic

specification does not specify the behavior of direct comparisons; these rules for comparisons involving a `NaN` were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the `compare()` and `compare-signal()` methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 Working with threads

The *getcontext()* function accesses a different *Context* object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext().prec=10`) without interfering with other threads.

Likewise, the *setcontext()* function automatically assigns its target to the current thread.

If *setcontext()* has not been called before *getcontext()*, then *getcontext()* will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called *DefaultContext*. To control the defaults so that each thread will use the same values throughout the application, directly modify the *DefaultContext* object. This should be done *before* any threads are started so that there won't be a race condition between threads calling *getcontext()*. For example:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 Recipes

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the *Decimal* class:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.
```

(continues on next page)

(continued from previous page)

```
    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383
```

(continues on next page)

(continued from previous page)

```
    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
```

(continues on next page)

(continued from previous page)

```
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.47942553860420300027328793521
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 Decimal FAQ

Q. It is cumbersome to type `decimal.Decimal('1234.5')`. Is there a way to minimize typing when using the interactive interpreter?

A. Some users abbreviate the constructor to just a single letter:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A. The `quantize()` method rounds to a fixed number of decimal places. If the *Inexact* trap is set, it is also useful for validation:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q. Once I have valid two place inputs, how do I maintain that invariant throughout an application?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a `quantize()` step:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

In developing fixed-point applications, it is convenient to define functions to handle the `quantize()` step:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                       # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. There are many ways to express the same value. The numbers `200`, `200.000`, `2E2`, and `02E+4` all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?

A. The `normalize()` method maps all equivalent values to a single representative:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing `5.0E+3` as `5000` keeps the value constant but cannot show the original's two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeroes, losing significance, but keeping the value unchanged:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. Is there a way to convert a regular float to a *Decimal*?

A. Yes, any binary floating point number can be exactly expressed as a Decimal though an exact conversion may take more precision than intuition would suggest:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. Within a complex calculation, how can I make sure that I haven't gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that "what you type is what you get". A disadvantage is that the results can look odd if you forget that the inputs haven't been rounded:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the *Context.create_decimal()* method:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

9.5 fractions — Rational numbers

Source code: Lib/fractions.py

The *fractions* module provides support for rational number arithmetic.

A Fraction instance can be constructed from a pair of integers, from another rational number, or from a string.

class fractions.Fraction(*numerator=0*, *denominator=1*)
class fractions.Fraction(*other_fraction*)
class fractions.Fraction(*float*)
class fractions.Fraction(*decimal*)
class fractions.Fraction(*string*)

The first version requires that *numerator* and *denominator* are instances of *numbers.Rational* and returns a new *Fraction* instance with value `numerator/denominator`. If *denominator* is `0`, it raises a *ZeroDivisionError*. The second version requires that *other_fraction* is an instance of *numbers.Rational* and returns a *Fraction* instance with the same value. The next two versions accept either a *float* or a *decimal.Decimal* instance, and return a *Fraction* instance with exactly the same value. Note that due to the usual issues with binary floating-point (see tut-fp-issues), the argument to `Fraction(1.1)` is not exactly equal to 11/10, and so `Fraction(1.1)` does *not* return `Fraction(11, 10)` as one might expect. (But see the documentation for the *limit_denominator()* method below.) The last version of the constructor expects a string or unicode instance. The usual form for this instance is:

```
[sign] numerator ['/' denominator]
```

where the optional `sign` may be either '+' or '-' and `numerator` and `denominator` (if present) are strings of decimal digits. In addition, any string that represents a finite value and is accepted by the *float* constructor is also accepted by the *Fraction* constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
```

(continues on next page)

(continued from previous page)

```
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

The *Fraction* class inherits from the abstract base class *numbers.Rational*, and implements all of the methods and operations from that class. *Fraction* instances are hashable, and should be treated as immutable. In addition, *Fraction* has the following properties and methods:

Changed in version 3.2: The *Fraction* constructor now accepts *float* and *decimal.Decimal* instances.

numerator
Numerator of the Fraction in lowest term.

denominator
Denominator of the Fraction in lowest term.

from_float(*flt*)
This class method constructs a *Fraction* representing the exact value of *flt*, which must be a *float*. Beware that `Fraction.from_float(0.3)` is not the same value as `Fraction(3, 10)`.

Note: From Python 3.2 onwards, you can also construct a *Fraction* instance directly from a *float*.

---

from_decimal(*dec*)
This class method constructs a *Fraction* representing the exact value of *dec*, which must be a *decimal.Decimal* instance.

Note: From Python 3.2 onwards, you can also construct a *Fraction* instance directly from a *decimal.Decimal* instance.

---

limit_denominator(*max_denominator=1000000*)
Finds and returns the closest *Fraction* to `self` that has denominator at most max_denominator. This method is useful for finding rational approximations to a given floating-point number:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

or for recovering a rational number that's represented as a float:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()
Returns the greatest *int* `<= self`. This method can also be accessed through the *math.floor()* function:

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()
: Returns the least *int* `>= self`. This method can also be accessed through the *math.ceil()* function.

__round__()
__round__(*ndigits*)
: The first version returns the nearest *int* to `self`, rounding half to even. The second version rounds `self` to the nearest multiple of `Fraction(1, 10**ndigits)` (logically, if `ndigits` is negative), again rounding half toward even. This method can also be accessed through the *round()* function.

fractions.gcd(*a*, *b*)
: Return the greatest common divisor of the integers *a* and *b*. If either *a* or *b* is nonzero, then the absolute value of `gcd(a, b)` is the largest integer that divides both *a* and *b*. `gcd(a,b)` has the same sign as *b* if *b* is nonzero; otherwise it takes the sign of *a*. `gcd(0, 0)` returns `0`.

  Deprecated since version 3.5: Use *math.gcd()* instead.

See also:

Module *numbers* The abstract base classes making up the numeric tower.

9.6 random — Generate pseudo-random numbers

Source code: Lib/random.py

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function *random()*, which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the `random.Random` class. You can instantiate your own instances of `Random` to get generators that don't share state.

Class `Random` can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the `random()`, `seed()`, `getstate()`, and `setstate()` methods. Optionally, a new generator can supply a `getrandbits()` method — this allows *randrange()* to produce selections over an arbitrarily large range.

The *random* module also provides the *SystemRandom* class which uses the system function *os.urandom()* to generate random numbers from sources provided by the operating system.

Warning: The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see the *secrets* module.

See also:

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.

9.6.1 Bookkeeping functions

`random.seed(`*a=None*, *version=2*`)`

Initialize the random number generator.

If *a* is omitted or `None`, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the *os.urandom()* function for details on availability).

If *a* is an int, it is used directly.

With version 2 (the default), a *str*, *bytes*, or *bytearray* object gets converted to an *int* and all of its bits are used.

With version 1 (provided for reproducing random sequences from older versions of Python), the algorithm for *str* and *bytes* generates a narrower range of seeds.

Changed in version 3.2: Moved to the version 2 scheme which uses all of the bits in a string seed.

`random.getstate()`

Return an object capturing the current internal state of the generator. This object can be passed to *setstate()* to restore the state.

`random.setstate(`*state*`)`

state should have been obtained from a previous call to *getstate()*, and *setstate()* restores the internal state of the generator to what it was at the time *getstate()* was called.

`random.getrandbits(`*k*`)`

Returns a Python integer with *k* random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, *getrandbits()* enables *randrange()* to handle arbitrarily large ranges.

9.6.2 Functions for integers

`random.randrange(`*stop*`)`

`random.randrange(`*start*, *stop*[, *step*]`)`

Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn't actually build a range object.

The positional argument pattern matches that of *range()*. Keyword arguments should not be used because the function may use them in unexpected ways.

Changed in version 3.2: *randrange()* is more sophisticated about producing equally distributed values. Formerly it used a style like `int(random()*n)` which could produce slightly uneven distributions.

`random.randint(`*a*, *b*`)`

Return a random integer *N* such that `a <= N <= b`. Alias for `randrange(a, b+1)`.

9.6.3 Functions for sequences

random.choice(*seq*)
Return a random element from the non-empty sequence *seq*. If *seq* is empty, raises *IndexError*.

random.choices(*population*, *weights=None*, ***, *cum_weights=None*, *k=1*)
Return a *k* sized list of elements chosen from the *population* with replacement. If the *population* is empty, raises *IndexError*.

If a *weights* sequence is specified, selections are made according to the relative weights. Alternatively, if a *cum_weights* sequence is given, the selections are made according to the cumulative weights (perhaps computed using *itertools.accumulate()*). For example, the relative weights `[10, 5, 30, 5]` are equivalent to the cumulative weights `[10, 15, 45, 50]`. Internally, the relative weights are converted to cumulative weights before making selections, so supplying the cumulative weights saves work.

If neither *weights* nor *cum_weights* are specified, selections are made with equal probability. If a weights sequence is supplied, it must be the same length as the *population* sequence. It is a *TypeError* to specify both *weights* and *cum_weights*.

The *weights* or *cum_weights* can use any numeric type that interoperates with the *float* values returned by *random()* (that includes integers, floats, and fractions but excludes decimals).

New in version 3.6.

random.shuffle(*x*[, *random*])
Shuffle the sequence *x* in place.

The optional argument *random* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function *random()*.

To shuffle an immutable sequence and return a new shuffled list, use `sample(x, k=len(x))` instead.

Note that even for small `len(x)`, the total number of permutations of *x* can quickly grow larger than the period of most random number generators. This implies that most permutations of a long sequence can never be generated. For example, a sequence of length 2080 is the largest that can fit within the period of the Mersenne Twister random number generator.

random.sample(*population*, *k*)
Return a *k* length list of unique elements chosen from the population sequence or set. Used for random sampling without replacement.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be *hashable* or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample from a range of integers, use a *range()* object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(range(10000000), k=60)`.

If the sample size is larger than the population size, a *ValueError* is raised.

9.6.4 Real-valued distributions

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution's equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

random.random()
Return the next random floating point number in the range [0.0, 1.0).

random.**uniform**(*a*, *b*)
Return a random floating point number *N* such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

The end-point value `b` may or may not be included in the range depending on floating-point rounding in the equation `a + (b-a) * random()`.

random.**triangular**(*low*, *high*, *mode*)
Return a random floating point number *N* such that `low <= N <= high` and with the specified *mode* between those bounds. The *low* and *high* bounds default to zero and one. The *mode* argument defaults to the midpoint between the bounds, giving a symmetric distribution.

random.**betavariate**(*alpha*, *beta*)
Beta distribution. Conditions on the parameters are `alpha > 0` and `beta > 0`. Returned values range between 0 and 1.

random.**expovariate**(*lambd*)
Exponential distribution. *lambd* is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called "lambda", but that is a reserved word in Python.) Returned values range from 0 to positive infinity if *lambd* is positive, and from negative infinity to 0 if *lambd* is negative.

random.**gammavariate**(*alpha*, *beta*)
Gamma distribution. (*Not* the gamma function!) Conditions on the parameters are `alpha > 0` and `beta > 0`.

The probability distribution function is:

```
            x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu*, *sigma*)
Gaussian distribution. *mu* is the mean, and *sigma* is the standard deviation. This is slightly faster than the *normalvariate()* function defined below.

random.**lognormvariate**(*mu*, *sigma*)
Log normal distribution. If you take the natural logarithm of this distribution, you'll get a normal distribution with mean *mu* and standard deviation *sigma*. *mu* can have any value, and *sigma* must be greater than zero.

random.**normalvariate**(*mu*, *sigma*)
Normal distribution. *mu* is the mean, and *sigma* is the standard deviation.

random.**vonmisesvariate**(*mu*, *kappa*)
mu is the mean angle, expressed in radians between 0 and 2**pi*, and *kappa* is the concentration parameter, which must be greater than or equal to zero. If *kappa* is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2**pi*.

random.**paretovariate**(*alpha*)
Pareto distribution. *alpha* is the shape parameter.

random.**weibullvariate**(*alpha*, *beta*)
Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.

9.6.5 Alternative Generator

class random.**SystemRandom**([*seed*])
Class that uses the *os.urandom()* function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state, and sequences are

not reproducible. Accordingly, the *seed()* method has no effect and is ignored. The *getstate()* and *setstate()* methods raise *NotImplementedError* if called.

9.6.6 Notes on Reproducibility

Sometimes it is useful to be able to reproduce the sequences given by a pseudo random number generator. By re-using a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

Most of the random module's algorithms and seeding functions are subject to change across Python versions, but two aspects are guaranteed not to change:

- If a new seeding method is added, then a backward compatible seeder will be offered.
- The generator's `random()` method will continue to produce the same sequence when the compatible seeder is given the same seed.

9.6.7 Examples and Recipes

Basic examples:

```
>>> random()                            # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                  # Random float:  2.5 <= x < 10.0
3.1800146073117523

>>> expovariate(1 / 5)                  # Interval between arrivals averaging 5 seconds
5.148957571865031

>>> randrange(10)                       # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)                # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])     # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                       # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)   # Four samples without replacement
[40, 10, 50, 30]
```

Simulations:

```
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck of 52 playing cards
```

(continues on next page)

(continued from previous page)

```
>>> # and determine the proportion of cards with a ten-value
>>> # (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = sample(list(deck.elements()), k=20)
>>> seen.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> trial = lambda: choices('HT', cum_weights=(0.60, 1.00), k=7).count('H') >= 5
>>> sum(trial() for i in range(10000)) / 10000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> trial = lambda : 2500 <= sorted(choices(range(10000), k=5))[2]  < 7500
>>> sum(trial() for i in range(10000)) / 10000
0.7958
```

Example of statistical bootstrapping using resampling with replacement to estimate a confidence interval for the mean of a sample of size five:

```
# http://statistics.about.com/od/Applications/a/Example-Of-Bootstrapping.htm
from statistics import mean
from random import choices

data = 1, 2, 4, 4, 10
means = sorted(mean(choices(data, k=5)) for i in range(20))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[1]:.1f} to {means[-2]:.1f}')
```

Example of a resampling permutation test to determine the statistical significance or p-value of an observed difference between the effects of a drug versus a placebo:

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

Simulation of arrival times and service deliveries in a single server queue:

```
from random import expovariate, gauss
from statistics import mean, median, stdev

average_arrival_interval = 5.6
average_service_time = 5.0
stdev_service_time = 0.5

num_waiting = 0
arrivals = []
starts = []
arrival = service_end = 0.0
for i in range(20000):
    if arrival <= service_end:
        num_waiting += 1
        arrival += expovariate(1.0 / average_arrival_interval)
        arrivals.append(arrival)
    else:
        num_waiting -= 1
        service_start = service_end if num_waiting else arrival
        service_time = gauss(average_service_time, stdev_service_time)
        service_end = service_start + service_time
        starts.append(service_start)

waits = [start - arrival for arrival, start in zip(arrivals, starts)]
print(f'Mean wait: {mean(waits):.1f}.  Stdev wait: {stdev(waits):.1f}.')
print(f'Median wait: {median(waits):.1f}.  Max wait: {max(waits):.1f}.')
```

See also:

Statistics for Hackers a video tutorial by Jake Vanderplas on statistical analysis using just a few fundamental concepts including simulation, sampling, shuffling, and cross-validation.

Economics Simulation a simulation of a marketplace by Peter Norvig that shows effective use of many of the tools and distributions provided by this module (gauss, uniform, sample, betavariate, choice, triangular, and randrange).

A Concrete Introduction to Probability (using Python) a tutorial by Peter Norvig covering the basics of probability theory, how to write simulations, and how to perform data analysis using Python.

9.7 `statistics` — Mathematical statistics functions

New in version 3.4.

Source code: Lib/statistics.py

This module provides functions for calculating mathematical statistics of numeric (`Real`-valued) data.

Note: Unless explicitly noted otherwise, these functions support *`int`*, *`float`*, *`decimal.Decimal`* and *`fractions.Fraction`*. Behaviour with other types (whether in the numeric tower or not) is currently unsupported. Mixed types are also undefined and implementation-dependent. If your input data consists of mixed types, you may be able to use *`map()`* to ensure a consistent result, e.g. `map(float, input_data)`.

9.7.1 Averages and measures of central location

These functions calculate an average or typical value from a population or sample.

mean()	Arithmetic mean ("average") of data.
harmonic_mean()	Harmonic mean of data.
median()	Median (middle value) of data.
median_low()	Low median of data.
median_high()	High median of data.
median_grouped()	Median, or 50th percentile, of grouped data.
mode()	Mode (most common value) of discrete data.

9.7.2 Measures of spread

These functions calculate a measure of how much the population or sample tends to deviate from the typical or average values.

pstdev()	Population standard deviation of data.
pvariance()	Population variance of data.
stdev()	Sample standard deviation of data.
variance()	Sample variance of data.

9.7.3 Function details

Note: The functions do not require the data given to them to be sorted. However, for reading convenience, most of the examples show sorted sequences.

`statistics.mean(`*data*`)`

Return the sample arithmetic mean of *data* which can be a sequence or iterator.

The arithmetic mean is the sum of the data divided by the number of data points. It is commonly called "the average", although it is only one of many different mathematical averages. It is a measure of the central location of the data.

If *data* is empty, *StatisticsError* will be raised.

Some examples of use:

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

Note: The mean is strongly affected by outliers and is not a robust estimator for central location: the mean is not necessarily a typical example of the data points. For more robust, although less efficient,

measures of central location, see *median()* and *mode()*. (In this case, "efficient" refers to statistical efficiency rather than computational efficiency.)

The sample mean gives an unbiased estimate of the true population mean, which means that, taken on average over all the possible samples, `mean(sample)` converges on the true mean of the entire population. If *data* represents the entire population rather than a sample, then `mean(data)` is equivalent to calculating the true population mean μ.

`statistics.harmonic_mean(`*data*`)`

Return the harmonic mean of *data*, a sequence or iterator of real-valued numbers.

The harmonic mean, sometimes called the subcontrary mean, is the reciprocal of the arithmetic *mean()* of the reciprocals of the data. For example, the harmonic mean of three values *a*, *b* and *c* will be equivalent to `3/(1/a + 1/b + 1/c)`.

The harmonic mean is a type of average, a measure of the central location of the data. It is often appropriate when averaging quantities which are rates or ratios, for example speeds. For example:

Suppose an investor purchases an equal value of shares in each of three companies, with P/E (price/earning) ratios of 2.5, 3 and 10. What is the average P/E ratio for the investor's portfolio?

```
>>> harmonic_mean([2.5, 3, 10])  # For an equal investment portfolio.
3.6
```

Using the arithmetic mean would give an average of about 5.167, which is too high.

StatisticsError is raised if *data* is empty, or any element is less than zero.

New in version 3.6.

`statistics.median(`*data*`)`

Return the median (middle value) of numeric data, using the common "mean of middle two" method. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

The median is a robust measure of central location, and is less affected by the presence of outliers in your data. When the number of data points is odd, the middle data point is returned:

```
>>> median([1, 3, 5])
3
```

When the number of data points is even, the median is interpolated by taking the average of the two middle values:

```
>>> median([1, 3, 5, 7])
4.0
```

This is suited for when your data is discrete, and you don't mind that the median may not be an actual data point.

If your data is ordinal (supports order operations) but not numeric (doesn't support addition), you should use *median_low()* or *median_high()* instead.

See also:

median_low(), *median_high()*, *median_grouped()*

`statistics.median_low(`*data*`)`

Return the low median of numeric data. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

The low median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the smaller of the two middle values is returned.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

Use the low median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

statistics.**median_high**(*data*)

Return the high median of data. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

The high median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the larger of the two middle values is returned.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

Use the high median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

statistics.**median_grouped**(*data*, *interval=1*)

Return the median of grouped continuous data, calculated as the 50th percentile, using interpolation. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

```
>>> median_grouped([52, 52, 53, 54])
52.5
```

In the following example, the data are rounded, so that each value represents the midpoint of data classes, e.g. 1 is the midpoint of the class 0.5–1.5, 2 is the midpoint of 1.5–2.5, 3 is the midpoint of 2.5–3.5, etc. With the data given, the middle value falls somewhere in the class 3.5–4.5, and interpolation is used to estimate it:

```
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 4, 5])
3.7
```

Optional argument *interval* represents the class interval, and defaults to 1. Changing the class interval naturally will change the interpolation:

```
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

This function does not check whether the data points are at least *interval* apart.

CPython implementation detail: Under some circumstances, *median_grouped()* may coerce data points to floats. This behaviour is likely to change in the future.

See also:

- "Statistics for the Behavioral Sciences", Frederick J Gravetter and Larry B Wallnau (8th Edition).
- The SSMEDIAN function in the Gnome Gnumeric spreadsheet, including this discussion.

statistics.**mode**(*data*)

Return the most common data point from discrete or nominal *data*. The mode (when it exists) is the most typical value, and is a robust measure of central location.

If *data* is empty, or if there is not exactly one most common value, *StatisticsError* is raised.

`mode` assumes discrete data, and returns a single value. This is the standard treatment of the mode as commonly taught in schools:

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

The mode is unique in that it is the only statistic which also applies to nominal (non-numeric) data:

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

statistics.**pstdev**(*data*, *mu=None*)

Return the population standard deviation (the square root of the population variance). See *pvariance()* for arguments and other details.

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.**pvariance**(*data*, *mu=None*)

Return the population variance of *data*, a non-empty iterable of real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *mu* is given, it should be the mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

Use this function to calculate the variance from the entire population. To estimate the variance from a sample, the *variance()* function is usually a better choice.

Raises *StatisticsError* if *data* is empty.

Examples:

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

If you have already calculated the mean of your data, you can pass it as the optional second argument *mu* to avoid recalculation:

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

This function does not attempt to verify that you have passed the actual mean as *mu*. Using arbitrary values for *mu* may lead to invalid or impossible results.

Decimals and Fractions are supported:

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')
```

(continues on next page)

(continued from previous page)

```
>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

Note: When called with the entire population, this gives the population variance σ^2. When called on a sample instead, this is the biased sample variance s^2, also known as variance with N degrees of freedom.

If you somehow know the true population mean μ, you may use this function to calculate the variance of a sample, giving the known population mean as the second argument. Provided the data points are representative (e.g. independent and identically distributed), the result will be an unbiased estimate of the population variance.

`statistics.stdev(`*data*, *xbar=None*`)`

Return the sample standard deviation (the square root of the sample variance). See *variance()* for arguments and other details.

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

`statistics.variance(`*data*, *xbar=None*`)`

Return the sample variance of *data*, an iterable of at least two real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *xbar* is given, it should be the mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

Use this function when your data is a sample from a population. To calculate the variance from the entire population, see *pvariance()*.

Raises *StatisticsError* if *data* has fewer than two values.

Examples:

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

If you have already calculated the mean of your data, you can pass it as the optional second argument *xbar* to avoid recalculation:

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

This function does not attempt to verify that you have passed the actual mean as *xbar*. Using arbitrary values for *xbar* can lead to invalid or impossible results.

Decimal and Fraction values are supported:

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
```

(continues on next page)

(continued from previous page)

```
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

Note: This is the sample variance s^2 with Bessel's correction, also known as variance with N-1 degrees of freedom. Provided that the data points are representative (e.g. independent and identically distributed), the result should be an unbiased estimate of the true population variance.

If you somehow know the actual population mean μ you should pass it to the *pvariance()* function as the *mu* parameter to get the variance of a sample.

9.7.4 Exceptions

A single exception is defined:

exception statistics.**StatisticsError**
Subclass of *ValueError* for statistics-related exceptions.