gmpy2 Documentation

Release 2.2.0a1

Case Van Horsen

Apr 06, 2023

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION TO GMPY2

gmpy2 is a C-coded Python extension module that supports multiple-precision arithmetic. gmpy2 is the successor to the original gmpy module. The gmpy module only supported the GMP multiple-precision library. gmpy2 adds support for the MPFR (correctly rounded real floating-point arithmetic) and MPC (correctly rounded complex floating-point arithmetic) libraries. gmpy2 also updates the API and naming conventions to be more consistent and support the additional functionality.

The following libraries are supported:

- GMP for integer and rational arithmetic

  Home page: http://gmplib.org

- MPFR for correctly rounded real floating-point arithmetic

  Home page: http://www.mpfr.org

- MPC for correctly rounded complex floating-point arithmetic

  Home page: http://mpc.multiprecision.org

- Generalized Lucas sequences and primality tests are based on the following code:

  mpz_lucas: http://sourceforge.net/projects/mpzlucas/

  mpz_prp: http://sourceforge.net/projects/mpzprp/

1.1 gmpy2 Versions

gmpy2 2.1.3 is the last planned release that will support Python 2.7 and the early Python 3 releases. Bugfixes may be released.

Development will shift to gmpy2 2.2.x

CHAPTER

TWO

INSTALLATION

Pre-compiled binary wheels are available on PyPI for Linux, MacOS, and Windows.

CHAPTER

THREE

OVERVIEW OF GMPY2

3.1 Tutorial

The *mpz* type is compatible with Python's built-in `int` type but is significantly faster for large values. The cutover point for performance varies, but can be as low as 20 to 40 digits. A variety of additional integer functions are provided.

Operator overloading is fully supported. Coversion from native Python types is optimized for performance.

```
>>> import gmpy2
>>> from gmpy2 import mpz,mpq,mpfr,mpc
>>> gmpy2.set_context(gmpy2.context())
>>> mpz(99) * 43
mpz(4257)
>>> pow(mpz(99), 37, 59)
mpz(18)
>>> gmpy2.isqrt(99)
mpz(9)
>>> gmpy2.isqrt_rem(99)
(mpz(9), mpz(18))
>>> gmpy2.gcd(123,27)
mpz(3)
>>> gmpy2.lcm(123,27)
mpz(1107)
>>> (mpz(123) + 12) / 5
mpfr('27.0')
>>> (mpz(123) + 12) // 5
mpz(27)
>>> (mpz(123) + 12) / 5.0
mpfr('27.0')
```

The *mpq* type is compatible with the `Fraction` type included with Python.

```
>>> mpq(3,7)/7
mpq(3,49)
>>> mpq(45,3) * mpq(11,8)
mpq(165,8)
```

gmpy2 supports correctly rounded arbitrary precision real and complex arithmetic via the MPFR and MPC libraries. Floating point contexts are used to control precision, rounding modes, and exceptional conditions. For example, division by zero can either return an Infinity or raise an exception.

```
>>> gmpy2.set_context(gmpy2.context())
>>> mpfr(1)/7
mpfr('0.14285714285714285')
>>> gmpy2.get_context().precision=200
>>> mpfr(1)/7
mpfr('0.1428571428571428571428571428571428571428571428571428571428571',200)
>>> gmpy2.get_context()
context(precision=200, real_prec=Default, imag_prec=Default,
        round=RoundToNearest, real_round=Default, imag_round=Default,
        emax=1073741823, emin=-1073741823,
        subnormalize=False,
        trap_underflow=False, underflow=False,
        trap_overflow=False, overflow=False,
        trap_inexact=False, inexact=True,
        trap_invalid=False, invalid=False,
        trap_erange=False, erange=False,
        trap_divzero=False, divzero=False,
        allow_complex=False,
        rational_division=False,
        allow_release_gil=False)
>>> mpfr(1)/0
mpfr('inf')
>>> gmpy2.get_context().trap_divzero=True
>>> mpfr(1)/0
Traceback (most recent call last):
...
gmpy2.DivisionByZeroError: division by zero
>>> gmpy2.get_context()
context(precision=200, real_prec=Default, imag_prec=Default,
        round=RoundToNearest, real_round=Default, imag_round=Default,
        emax=1073741823, emin=-1073741823,
        subnormalize=False,
        trap_underflow=False, underflow=False,
        trap_overflow=False, overflow=False,
        trap_inexact=False, inexact=True,
        trap_invalid=False, invalid=False,
        trap_erange=False, erange=False,
        trap_divzero=True, divzero=True,
        allow_complex=False,
        rational_division=False,
        allow_release_gil=False)
>>> gmpy2.sqrt(mpfr(-2))
mpfr('nan')
>>> gmpy2.get_context().allow_complex=True
>>> gmpy2.get_context().precision=53
>>> gmpy2.sqrt(mpfr(-2))
mpc('0.0+1.4142135623730951j')
>>>
>>> gmpy2.set_context(gmpy2.context())
>>> with gmpy2.local_context() as ctx:
...     print(gmpy2.const_pi())
...     ctx.precision+=20
```

(continues on next page)

(continued from previous page)

```
...   print(gmpy2.const_pi())
...   ctx.precision+=20
...   print(gmpy2.const_pi())
...
3.1415926535897931
3.141592653589793238462643383l
3.14159265358979323846264338327950288l
>>> print(gmpy2.const_pi())
3.1415926535897931
```

3.2 Miscellaneous gmpy2 Functions

gmpy2.**digits**(*x*, *base=10*, *prec=0*, */*) → str

Return string representing a number x.

gmpy2.**from_binary**(*bytes*, */*) → *mpz* | *xmpz* | *mpq* | *mpfr* | *mpc*

Return a Python object from a byte sequence created by *to_binary()*.

gmpy2.**license**() → str

Return string giving license information.

gmpy2.**mp_limbsize**() → int

Return the number of bits per limb.

gmpy2.**mp_version**() → str

Return string giving current GMP version.

gmpy2.**mpc_version**() → str

Return string giving current MPC version.

gmpy2.**mpfr_version**() → str

Return string giving current MPFR version.

gmpy2.**random_state**(*seed=0*, */*) → object

Return new object containing state information for the random number generator. An optional integer can be specified as the seed value.

gmpy2.**to_binary**(*x*, */*) → bytes

Return a Python byte sequence that is a portable binary representation of a gmpy2 object x. The byte sequence can be passed to *from_binary()* to obtain an exact copy of x's value. Raises a TypeError if x is not a gmpy2 object.

gmpy2.**version**() → str

Return string giving current GMPY2 version.

3.3 Generic gmpy2 Functions

gmpy2.**add**(*x*, *y*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return x + y.

gmpy2.**div**(*x*, *y*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return x / y; uses true division.

gmpy2.**mul**(*x*, *y*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return x * y.

gmpy2.**sub**(*x*, *y*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return x - y.

gmpy2.**square**(*x*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return x * x.

gmpy2.**f2q**(*x*, *err=0*, /) → *mpz* | *mpq*

Return the 'best' *mpq* approximating x to within relative error err. Default is the precision of x. Uses Stern-Brocot tree to find the 'best' approximation. An *mpz* object is returned if the denominator is 1. If err<0, relative error is 2.0 ** err.

gmpy2.**fma**(*x*, *y*, *z*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return correctly rounded result of (x * y) + z.

gmpy2.**fms**(*x*, *y*, *z*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return correctly rounded result of (x * y) - z.

gmpy2.**cmp_abs**(*x*, *y*, /) → int

Return -1 if abs(x) < abs(y); 0 if abs(x) = abs(y); or 1 else.

3.4 Exceptions

exception gmpy2.**RangeError**

exception gmpy2.**InexactResultError**

exception gmpy2.**OverflowResultError**

exception gmpy2.**UnderflowResultError**

exception gmpy2.**InvalidOperationError**

exception gmpy2.**DivisionByZeroError**

CHAPTER

FOUR

MULTIPLE-PRECISION INTEGERS

The gmpy2 *mpz* type supports arbitrary precision integers. It should be a drop-in replacement for Python's `int` type. Depending on the platform and the specific operation, an *mpz* will be faster than Python's `int` once the precision exceeds 20 to 50 digits. All the special integer functions in GMP are supported.

4.1 Examples

```
>>> from gmpy2 import is_prime, mpz
>>> mpz('123') + 1
mpz(124)
>>> 10 - mpz(1)
mpz(9)
>>> is_prime(17)
True
>>> mpz('1_2')
mpz(12)
```

Note: The use of `from gmpy2 import *` is not recommended. The names in gmpy2 have been chosen to avoid conflict with Python's builtin names but gmpy2 does use names that may conflict with other modules or variable names.

Note: *mpz* ignores all embedded underscore characters. It does not attempt to be 100% compatible with all Python exceptions.

4.2 mpz type

class gmpy2.**mpz**(*n=0*, */*)

class gmpy2.**mpz**(*s*, */*, *base=0*)

Return an immutable integer constructed from a numeric value n (truncating n to its integer part) or a string s made of digits in the given base. Every input, that is accepted by the `int` type constructor is also accepted.

The base may vary from 2 to 62, or if base is 0, then binary, octal, or hexadecimal strings are recognized by leading '0b', '0o', or '0x' characters (case is ignored), otherwise the string is assumed to be decimal. For bases up to 36, digits case is ignored. For bases 37 to 62, upper-case letter represent the usual 10..35 range, while lower-case letter represent 36..61. Optionally the string can be preceded by '+' or '-'. White space and underscore is simply ignored.

__format__(*fmt*) → str

Return a Python string by formatting *mpz* 'x' using the format string 'fmt'. A valid format string consists of:

optional alignment code:

'<' -> left shifted in field '>' -> right shifted in field '^' -> centered in field

optional leading sign code:

'+' -> always display leading sign '-' -> only display minus sign ' ' -> minus for negative values, space for positive values

optional base indicator

'#' -> precede binary, octal, or hex with 0b, 0o or 0x

optional width

optional conversion code:

'd' -> decimal format 'b' -> binary format 'o' -> octal format 'x' -> hex format 'X' -> upper-case hex format

The default format is 'd'.

as_integer_ratio() → tuple[*mpz*, *mpz*]

Return a pair of integers, whose ratio is exactly equal to the original number. The ratio is in lowest terms and has a positive denominator.

bit_clear(*n*, /) → *mpz*

Return a copy of x with the n-th bit cleared.

bit_count() → int

Return the number of 1-bits set in abs(x).

bit_flip(*n*, /) → *mpz*

Return a copy of x with the n-th bit inverted.

bit_length() → int

Return the number of significant bits in the radix-2 representation of x. Note: mpz(0).bit_length() returns 0.

bit_scan0(*n=0*, /) → int | None

Return the index of the first 0-bit of x with index >= n. n >= 0. If there are no more 0-bits in x at or above index n (which can only happen for x<0, assuming an infinitely long 2's complement format), then `None` is returned.

bit_scan1(*n=0*, /) → int | None

Return the index of the first 1-bit of x with index >= n. n >= 0. If there are no more 1-bits in x at or above index n (which can only happen for x>=0, assuming an infinitely long 2's complement format), then `None` is returned.

bit_set(*n*, /) → *mpz*

Return a copy of x with the n-th bit set.

bit_test(*n*, /) → bool

Return the value of the n-th bit of x.

conjugate() → *mpz*

Return the conjugate of x (which is just a new reference to x since x is not a complex number).

digits(*base=10*, */*) → str

Return Python string representing x in the given base. Values for base can range between 2 to 62. A leading '-' is present if x<0 but no leading '+' is present if x>=0.

from_bytes(*bytes*, *byteorder='big'*, ***, *signed=False*) → *mpz*

Return the integer represented by the given array of bytes.

bytes
Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. `bytes` and `bytearray` are examples of built-in objects that support the buffer protocol.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

signed
Indicates whether two's complement is used to represent the integer.

is_congruent(*y*, *m*, */*) → bool

Returns `True` if x is congruent to y modulo m, else return `False`.

is_divisible(*d*, */*) → bool

Returns `True` if x is divisible by d, else return `False`.

is_even() → bool

Return `True` if x is even, `False` otherwise.

is_odd() → bool

Return `True` if x is odd, `False` otherwise.

is_power() → bool

Return `True` if x is a perfect power (there exists a y and an n > 1, such that x=y**n), else return `False`.

is_prime(*n=25*, */*) → bool

Return `True` if x is _probably_ prime, else `False` if x is definitely composite. x is checked for small divisors and up to n Miller-Rabin tests are performed.

is_probab_prime(*n=25*, */*) → int

Return 2 if x is definitely prime, 1 if x is probably prime, or return 0 if x is definitely non-prime. x is checked for small divisors and up to n Miller-Rabin tests are performed. Reasonable values of n are between 15 and 50.

is_square() → bool

Returns `True` if x is a perfect square, else return `False`.

num_digits(*base=10*, */*) → int

Return length of string representing the absolute value of x in the given base. Values for base can range between 2 and 62. The value returned may be 1 too large.

to_bytes(*length=1*, *byteorder='big'*, ***, *signed=False*) → bytes

Return an array of bytes representing an integer.

length
Length of bytes object to use. An `OverflowError` is raised if the integer is not representable with the given number of bytes.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

signed
Determines whether two's complement is used to represent the integer. If signed is `False` and a negative integer is given, an `OverflowError` is raised.

denominator
the denominator of a rational number in lowest terms

imag
the imaginary part of a complex number

numerator
the numerator of a rational number in lowest terms

real
the real part of a complex number

4.3 mpz Functions

gmpy2.**bincoef**(*n*, *k*, /) → *mpz*
Return the binomial coefficient ('n choose k'). k >= 0.

gmpy2.**bit_clear**(*x*, *n*, /) → *mpz*
Return a copy of x with the n-th bit cleared.

gmpy2.**bit_count**(*x*, /) → int
Return the number of 1-bits set in abs(x).

gmpy2.**bit_flip**(*x*, *n*, /) → *mpz*
Return a copy of x with the n-th bit inverted.

gmpy2.**bit_length**(*x*, /) → int
Return the number of significant bits in the radix-2 representation of x. Note: bit_length(0) returns 0.

gmpy2.**bit_mask**(*n*, /) → *mpz*
Return an *mpz* exactly n bits in length with all bits set.

gmpy2.**bit_scan0**(*x*, *n=0*, /) → int | None
Return the index of the first 0-bit of x with index >= n. n >= 0. If there are no more 0-bits in x at or above index n (which can only happen for x<0, assuming an infinitely long 2's complement format), then `None` is returned.

gmpy2.**bit_scan1**(*x*, *n=0*, /) → int | None
Return the index of the first 1-bit of x with index >= n. n >= 0. If there are no more 1-bits in x at or above index n (which can only happen for x>=0, assuming an infinitely long 2's complement format), then `None` is returned.

gmpy2.**bit_set**(*x*, *n*, /) → *mpz*
Return a copy of x with the n-th bit set.

gmpy2.**bit_test**(*x*, *n*, /) → bool
Return the value of the n-th bit of x.

gmpy2.**c_div**(*x*, *y*, /) → *mpz*

Return the quotient of x divided by y. The quotient is rounded towards +Inf (ceiling rounding). x and y must be integers.

gmpy2.**c_div_2exp**(*x*, *n*, /) → *mpz*

Returns the quotient of x divided by 2**n. The quotient is rounded towards +Inf (ceiling rounding). x must be an integer. n must be >0.

gmpy2.**c_divmod**(*x*, *y*, /) → tuple[*mpz*, *mpz*]

Return the quotient and remainder of x divided by y. The quotient is rounded towards +Inf (ceiling rounding) and the remainder will have the opposite sign of y. x and y must be integers.

gmpy2.**c_divmod_2exp**(*x*, *n*, /) → tuple[*mpz*, *mpz*]

Return the quotient and remainder of x divided by 2**n. The quotient is rounded towards +Inf (ceiling rounding) and the remainder will be negative. x must be an integer. n must be >0.

gmpy2.**c_mod**(*x*, *y*, /) → *mpz*

Return the remainder of x divided by y. The remainder will have the opposite sign of y. x and y must be integers.

gmpy2.**c_mod_2exp**(*x*, *n*, /) → *mpz*

Return the remainder of x divided by 2**n. The remainder will be negative. x must be an integer. n must be >0.

gmpy2.**comb**(*n*, *k*, /) → *mpz*

Return the number of combinations of 'n things, taking k at a time'. k >= 0. Same as bincoef(n, k)

gmpy2.**divexact**(*x*, *y*, /) → *mpz*

Return the quotient of x divided by y. Faster than standard division but requires the remainder is zero!

gmpy2.**divm**(*a*, *b*, *m*, /) → *mpz*

Return x such that b*x == a mod m. Raises a `ZeroDivisionError` exception if no such value x exists.

gmpy2.**double_fac**(*n*, /) → *mpz*

Return the exact double factorial (n!!) of n. The double factorial is defined as n*(n-2)*(n-4)...

gmpy2.**f_div**(*x*, *y*, /) → *mpz*

Return the quotient of x divided by y. The quotient is rounded towards -Inf (floor rounding). x and y must be integers.

gmpy2.**f_div_2exp**(*x*, *n*, /) → *mpz*

Return the quotient of x divided by 2**n. The quotient is rounded towards -Inf (floor rounding). x must be an integer. n must be >0.

gmpy2.**f_divmod**(*x*, *y*, /) → tuple[*mpz*, *mpz*]

Return the quotient and remainder of x divided by y. The quotient is rounded towards -Inf (floor rounding) and the remainder will have the same sign as y. x and y must be integers.

gmpy2.**f_divmod_2exp**(*x*, *n*, /) → tuple[*mpz*, *mpz*]

Return quotient and remainder after dividing x by 2**n. The quotient is rounded towards -Inf (floor rounding) and the remainder will be positive. x must be an integer. n must be >0.

gmpy2.**f_mod**(*x*, *y*, /) → *mpz*

Return the remainder of x divided by y. The remainder will have the same sign as y. x and y must be integers.

gmpy2.**f_mod_2exp**(*x*, *n*, /) → *mpz*

Return remainder of x divided by 2**n. The remainder will be positive. x must be an integer. n must be >0.

gmpy2.**fac**(*n*, /) → *mpz*

Return the exact factorial of n.

See factorial(n) to get the floating-point approximation.

gmpy2.**fib**(*n*, /) → *mpz*

Return the n-th Fibonacci number.

gmpy2.**fib2**(*n*, /) → tuple[*mpz*, *mpz*]

Return a 2-tuple with the (n-1)-th and n-th Fibonacci numbers.

gmpy2.**gcd**(**integers*, /) → *mpz*

Return the greatest common divisor of integers.

gmpy2.**gcdext**(*a*, *b*, /) → tuple[*mpz*, *mpz*, *mpz*]

Return a 3-element tuple (g,s,t) such that g == gcd(a,b) and g == a*s + b*t.

gmpy2.**hamdist**(*x*, *y*, /) → int

Return the Hamming distance (number of bit-positions where the bits differ) between integers x and y.

gmpy2.**invert**(*x*, *m*, /) → *mpz*

Return y such that x*y == 1 modulo m. Raises `ZeroDivisionError` if no inverse exists.

gmpy2.**iroot**(*x*, *n*, /) → tuple[*mpz*, bool]

Return the integer n-th root of x and boolean value that is `True` iff the root is exact. x >= 0. n > 0.

gmpy2.**iroot_rem**(*x*, *n*, /) → tuple[*mpz*, *mpz*]

Return a 2-element tuple (y,r), such that y is the integer n-th root of x and x=y**n + r. x >= 0. n > 0.

gmpy2.**is_congruent**(*x*, *y*, *m*, /) → bool

Returns `True` if x is congruent to y modulo m, else return `False`.

gmpy2.**is_divisible**(*x*, *d*, /) → bool

Returns `True` if x is divisible by d, else return `False`.

gmpy2.**is_even**(*x*, /) → bool

Return `True` if x is even, `False` otherwise.

gmpy2.**is_odd**(*x*, /) → bool

Return `True` if x is odd, `False` otherwise.

gmpy2.**is_power**(*x*, /) → bool

Return `True` if x is a perfect power (there exists a y and an n > 1, such that x=y**n), else return `False`.

gmpy2.**is_prime**(*x*, *n=25*, /) → bool

Return `True` if x is _probably_ prime, else `False` if x is definitely composite. x is checked for small divisors and up to n Miller-Rabin tests are performed.

gmpy2.**is_probab_prime**(*x*, *n=25*, /) → int

Return 2 if x is definitely prime, 1 if x is probably prime, or return 0 if x is definitely non-prime. x is checked for small divisors and up to n Miller-Rabin tests are performed. Reasonable values of n are between 15 and 50.

gmpy2.**is_square**(*x*, /) → bool

Returns `True` if x is a perfect square, else return `False`.

gmpy2.**isqrt**(*x*, /) → *mpz*

Return the integer square root of a non-negative integer x.

gmpy2.**isqrt_rem**(*x*, /)

Return a 2-element tuple (s,t) such that s=isqrt(x) and t=x-s*s. x >=0.

gmpy2.**jacobi**(*x*, *y*, /) → *mpz*

Return the Jacobi symbol (x|y). y must be odd and >0.

gmpy2.**kronecker**(*x*, *y*, /) → *mpz*

Return the Kronecker-Jacobi symbol (x|y).

gmpy2.**lcm**(**integers*, /) → *mpz*

Return the lowest common multiple of integers.

gmpy2.**legendre**(*x*, *y*, /) → *mpz*

Return the Legendre symbol (x|y). y is assumed to be an odd prime.

gmpy2.**lucas**(*n*, /) → *mpz*

Return the n-th Lucas number.

gmpy2.**lucas2**(*n*, /) → tuple[*mpz*, *mpz*]

Return a 2-tuple with the (n-1)-th and n-th Lucas numbers.

gmpy2.**mpz_random**(*random_state*, *int*, /) → *mpz*

Return uniformly distributed random integer between 0 and n-1.

gmpy2.**mpz_rrandomb**(*random_state*, *bit_count*, /) → *mpz*

Return a random integer between 0 and 2**bit_count-1 with long sequences of zeros and one in its binary representation.

gmpy2.**mpz_urandomb**(*random_state*, *bit_count*, /) → *mpz*

Return uniformly distributed random integer between 0 and 2**bit_count-1.

gmpy2.**multi_fac**(*n*, *m*, /) → *mpz*

Return the exact m-multi factorial of n. The m-multifactorial is defined as n*(n-m)*(n-2m)...

gmpy2.**next_prime**(*x*, /) → *mpz*

Return the next *probable* prime number > x.

gmpy2.**num_digits**(*x*, *base=10*, /) → int

Return length of string representing the absolute value of x in the given base. Values for base can range between 2 and 62. The value returned may be 1 too large.

gmpy2.**popcount**(*x*, /) → int

Return the number of 1-bits set in x. If x<0, the number of 1-bits is infinite so -1 is returned in that case.

gmpy2.**powmod**(*x*, *y*, *m*, /) → *mpz*

Return (x**y) mod m. Same as the three argument version of Python's built-in pow, but converts all three arguments to *mpz*.

gmpy2.**powmod_exp_list**(*base*, *exp_lst*, *mod*, /) → list[*mpz*, ...]

Returns list(powmod(base, i, mod) for i in exp_lst). Will always release the GIL. (Experimental in gmpy2 2.1.x).

gmpy2.**powmod_base_list**(*base_lst*, *exp*, *mod*, /) → list[*mpz*, ...]

Returns list(powmod(i, exp, mod) for i in base_lst). Will always release the GIL. (Experimental in gmpy2 2.1.x).

gmpy2.**powmod_sec**(*x*, *y*, *m*, /) → *mpz*

Return (x**y) mod m. Calculates x ** y (mod m) but using a constant time algorithm to reduce the risk of side channel attacks. y must be an integer >0. m must be an odd integer.

gmpy2.**primorial**(*n*, /) → *mpz*

Return the product of all positive prime numbers less than or equal to n.

gmpy2.**remove**(*x*, *f*, /) → tuple[*mpz*, *mpz*]

Return a 2-element tuple (y,m) such that x=y*(f**m) and f does not divide y. Remove the factor f from x as many times as possible. m is the multiplicity f in x. f > 1.

gmpy2.**t_div**(*x*, *y*, /) → *mpz*

Return the quotient of x divided by y. The quotient is rounded towards 0. x and y must be integers.

gmpy2.**t_div_2exp**(*x*, *n*, /) → *mpz*

Return the quotient of x divided by 2**n. The quotient is rounded towards zero (truncation). n must be >0.

gmpy2.**t_divmod**(*x*, *y*, /) → tuple[*mpz*, *mpz*]

Return the quotient and remainder of x divided by y. The quotient is rounded towards zero (truncation) and the remainder will have the same sign as x. x and y must be integers.

gmpy2.**t_divmod_2exp**(*x*, *n*, /) → tuple[*mpz*, *mpz*]

Return the quotient and remainder of x divided by 2**n. The quotient is rounded towards zero (truncation) and the remainder will have the same sign as x. x must be an integer. n must be >0.

gmpy2.**t_mod**(*x*, *y*, /) → *mpz*

Return the remainder of x divided by y. The remainder will have the same sign as x. x and y must be integers.

gmpy2.**t_mod_2exp**(*x*, *n*, /) → *mpz*

Return the remainder of x divided by 2**n. The remainder will have the same sign as x. x must be an integer. n must be >0.

CHAPTER

FIVE

MULTIPLE-PRECISION INTEGERS (ADVANCED TOPICS)

5.1 The xmpz type

gmpy2 provides access to an experimental integer type called *xmpz*. The *xmpz* type is a mutable integer type. In-place operations (+=, //=, etc.) modify the original object and do not create a new object. Instances of *xmpz* cannot be used as dictionary keys.

```
>>> from gmpy2 import xmpz
>>> a = xmpz(123)
>>> b = a
>>> a += 1
>>> a
xmpz(124)
>>> b
xmpz(124)
```

The ability to change an *xmpz* object in-place allows for efficient and rapid bit manipulation.

Individual bits can be set or cleared:

```
>>> a[10]=1
>>> a
xmpz(1148)
```

Slice notation is supported. The bits referenced by a slice can be either 'read from' or 'written to'. To clear a slice of bits, use a source value of 0. In 2s-complement format, 0 is represented by an arbitrary number of 0-bits. To set a slice of bits, use a source value of ~0. The *tilde* operator inverts, or complements the bits in an integer. (~0 is -1 so you can also use -1.) In 2s-complement format, -1 is represented by an arbitrary number of 1-bits.

If a value for *stop* is specified in a slice assignment and the actual bit-length of the *xmpz* is less than *stop*, then the destination *xmpz* is logically padded with 0-bits to length *stop*.

```
>>> a=xmpz(0)
>>> a[8:16] = ~0
>>> bin(a)
'0b1111111100000000'
>>> a[4:12] = ~a[4:12]
>>> bin(a)
'0b1111000011110000'
```

Bits can be reversed:

```
>>> a = xmpz(1148)
>>> bin(a)
'0b10001111100'
>>> a[::] = a[::-1]
>>> bin(a)
'0b111110001'
```

The *iter_bits()* method returns a generator that returns True or False for each bit position. The methods *iter_clear()*, and *iter_set()* return generators that return the bit positions that are 1 or 0. The methods support arguments *start* and *stop* that define the beginning and ending bit positions that are used. To mimic the behavior of slices. the bit positions checked include *start* but the last position checked is *stop* - 1.

```
>>> a=xmpz(117)
>>> bin(a)
'0b1110101'
>>> list(a.iter_bits())
[True, False, True, False, True, True, True]
>>> list(a.iter_clear())
[1, 3]
>>> list(a.iter_set())
[0, 2, 4, 5, 6]
>>> list(a.iter_bits(stop=12))
[True, False, True, False, True, True, True, False, False, False, False, False]
```

The following program uses the Sieve of Eratosthenes to generate a list of prime numbers.

```
import time
import gmpy2

def sieve(limit=1000000):
    "Returns a generator that yields the prime numbers up to limit."

    # Increment by 1 to account for the fact that slices  do not include
    # the last index value but we do want to include the last value for
    # calculating a list of primes.
    sieve_limit = gmpy2.isqrt(limit) + 1
    limit += 1

    # Mark bit positions 0 and 1 as not prime.
    bitmap = gmpy2.xmpz(3)

    # Process 2 separately. This allows us to use p+p for the step size
    # when sieving the remaining primes.
    bitmap[4 : limit : 2] = -1

    # Sieve the remaining primes.
    for p in bitmap.iter_clear(3, sieve_limit):
        bitmap[p*p : limit : p+p] = -1

    return bitmap.iter_clear(2, limit)

if __name__ == "__main__":
    start = time.time()
```

(continues on next page)

(continued from previous page)

```
    result = list(sieve())
    print(time.time() - start)
    print(len(result))
```

class gmpy2.**xmpz**(*n=0*, */*)

class gmpy2.**xmpz**(*s*, */*, *base=0*)

Return a mutable integer constructed from a numeric value n or a string s made of digits in the given base. Every input, that is accepted by the *mpz* type constructor is also accepted.

Note: This type can be faster when used for augmented assignment (+=, -=, etc), but *xmpz* objects cannot be used as dictionary keys.

__format__(*fmt*) → str

Return a Python string by formatting *mpz* 'x' using the format string 'fmt'. A valid format string consists of:

optional alignment code:

'<' -> left shifted in field '>' -> right shifted in field '^' -> centered in field

optional leading sign code:

'+' -> always display leading sign '-' -> only display minus sign ' ' -> minus for negative values, space for positive values

optional base indicator

'#' -> precede binary, octal, or hex with 0b, 0o or 0x

optional width

optional conversion code:

'd' -> decimal format 'b' -> binary format 'o' -> octal format 'x' -> hex format 'X' -> upper-case hex format

The default format is 'd'.

bit_clear(*n*, */*) → *mpz*

Return a copy of x with the n-th bit cleared.

bit_flip(*n*, */*) → *mpz*

Return a copy of x with the n-th bit inverted.

bit_length() → int

Return the number of significant bits in the radix-2 representation of x. Note: mpz(0).bit_length() returns 0.

bit_scan0(*n=0*, */*) → int | None

Return the index of the first 0-bit of x with index >= n. n >= 0. If there are no more 0-bits in x at or above index n (which can only happen for x<0, assuming an infinitely long 2's complement format), then `None` is returned.

bit_scan1(*n=0*, */*) → int | None

Return the index of the first 1-bit of x with index >= n. n >= 0. If there are no more 1-bits in x at or above index n (which can only happen for x>=0, assuming an infinitely long 2's complement format), then `None` is returned.

bit_set(*n*, /) → *mpz*

Return a copy of x with the n-th bit set.

bit_test(*n*, /) → bool

Return the value of the n-th bit of x.

conjugate() → *mpz*

Return the conjugate of x (which is just a new reference to x since x is not a complex number).

copy() → *xmpz*

Return a copy of a x.

digits(*base=10*, /) → str

Return Python string representing x in the given base. Values for base can range between 2 to 62. A leading '-' is present if x<0 but no leading '+' is present if x>=0.

iter_bits(*start=0*, *stop=-1*) → collections.abc.Iterator

Return `True` or `False` for each bit position in x beginning at 'start'. If a positive value is specified for 'stop', iteration is continued until 'stop' is reached. If a negative value is specified, iteration is continued until the last 1-bit. Note: the value of the underlying *xmpz* object can change during iteration.

iter_clear(*start=0*, *stop=-1*) → collections.abc.Iterator

Return every bit position that is clear in x, beginning at 'start'. If a positive value is specified for 'stop', iteration is continued until 'stop' is reached. If a negative value is specified, iteration is continued until the last 1-bit. Note: the value of the underlying *xmpz* object can change during iteration.

iter_set(*start=0*, *stop=-1*) → collections.abc.Iterator

Return an iterator yielding the bit position for every bit that is set in x, beginning at 'start'. If a positive value is specified for 'stop', iteration is continued until 'stop' is reached. To match the behavior of slicing, 'stop' is not included. If a negative value is specified, iteration is continued until the last 1-bit. Note: the value of the underlying *xmpz* object can change during iteration.

limbs_finish(*n*, /) → None

Must be called after writing to the address returned by x.limbs_write(n) or x.limbs_modify(n) to update the limbs of x.

limbs_modify(*n*, /) → int

Returns the address of a mutable buffer representing the limbs of x, resized so that it may hold at least n limbs. Must be followed by a call to x.limbs_finish(n) after writing to the returned address in order for the changes to take effect.

limbs_read() → int

Returns the address of the immutable buffer representing the limbs of x.

limbs_write(*n*, /) → int

Returns the address of a mutable buffer representing the limbs of x, resized so that it may hold at least n limbs. Must be followed by a call to x.limbs_finish(n) after writing to the returned address in order for the changes to take effect. WARNING: this operation is destructive and may destroy the old value of x.

make_mpz() → *mpz*

Return an *mpz* by converting x as quickly as possible.

NOTE: Optimized for speed so the original *xmpz* value is set to 0!

num_digits(*base=10*, /) → int

Return length of string representing the absolute value of x in the given base. Values for base can range between 2 and 62. The value returned may be 1 too large.

num_limbs() → int

Return the number of limbs of x.

denominator

the denominator of a rational number in lowest terms

numerator

the numerator of a rational number in lowest terms

real

the real part of a complex number

5.2 Advanced Number Theory Functions

The following functions are based on mpz_lucas.c and mpz_prp.c by David Cleaver.

A good reference for probable prime testing is http://www.pseudoprime.com/pseudo.html

gmpy2.**is_bpsw_prp**(*n*, /) → bool

Return True if n is a Baillie-Pomerance-Selfridge-Wagstaff probable prime. A BPSW probable prime passes the *is_strong_prp()* test with base 2 and the *is_selfridge_prp()* test.

gmpy2.**is_euler_prp**(*n*, *a*, /) → bool

Return True if n is an Euler (also known as Solovay-Strassen) probable prime to the base a. Assuming:

gcd(n,a) == 1 n is odd

Then an Euler probable prime requires:

a**((n-1)/2) == 1 (mod n)

gmpy2.**is_extra_strong_lucas_prp**(*n*, *p*, /) → bool

Return True if n is an extra strong Lucas probable prime with parameters (p,1). Assuming:

n is odd D = p*p - 4, D != 0 gcd(n, 2*D) == 1 n = s*(2**r) + Jacobi(D,n), s odd

Then an extra strong Lucas probable prime requires:

lucasu(p,1,s) == 0 (mod n) or lucasv(p,1,s) == +/-2 (mod n) or lucasv(p,1,s*(2**t)) == 0 (mod n) for some t, 0 <= t < r

gmpy2.**is_fermat_prp**(*n*, *a*, /) → bool

Return True if n is a Fermat probable prime to the base a. Assuming:

gcd(n,a) == 1

Then a Fermat probable prime requires:

a**(n-1) == 1 (mod n)

gmpy2.**is_fibonacci_prp**(*n*, *p*, *q*, /) → bool

Return True if n is a Fibonacci probable prime with parameters (p,q). Assuming:

n is odd p > 0, q = +/-1 p*p - 4*q != 0

Then a Fibonacci probable prime requires:

lucasv(p,q,n) == p (mod n).

gmpy2.**is_lucas_prp**(*n*, *p*, *q*, /) → bool

Return True if n is a Lucas probable prime with parameters (p,q). Assuming:

n is odd D = p*p - 4*q, D != 0 gcd(n, 2*q*D) == 1

Then a Lucas probable prime requires:

lucasu(p,q,n - Jacobi(D,n)) == 0 (mod n)

gmpy2.**is_selfridge_prp**(*n*, /) → bool

Return True if n is a Lucas probable prime with Selfidge parameters (p,q). The Selfridge parameters are chosen by finding the first element D in the sequence {5, -7, 9, -11, 13, ...} such that Jacobi(D,n) == -1. Then let p=1 and q = (1-D)/4. Then perform a Lucas probable prime test.

gmpy2.**is_strong_bpsw_prp**(*n*, /) → bool

Return True if n is a strong Baillie-Pomerance-Selfridge-Wagstaff probable prime. A strong BPSW probable prime passes the *is_strong_prp()* test with base and the *is_strong_selfridge_prp()* test.

gmpy2.**is_strong_lucas_prp**(*n*, *p*, *q*, /) → bool

Return True if n is a strong Lucas probable prime with parameters (p,q). Assuming:

n is odd D = p*p - 4*q, D != 0 gcd(n, 2*q*D) == 1 n = s*(2**r) + Jacobi(D,n), s odd

Then a strong Lucas probable prime requires:

lucasu(p,q,s) == 0 (mod n) or lucasv(p,q,s*(2**t)) == 0 (mod n) for some t, 0 <= t < r

gmpy2.**is_strong_prp**(*n*, *a*, /) → bool

Return True if n is a strong (also known as Miller-Rabin) probable prime to the base a. Assuming:

gcd(n,a) == 1 n is odd n = s*(2**r) + 1, with s odd

Then a strong probable prime requires one of the following is true:

a**s == 1 (mod n) or a**(s*(2**t)) == -1 (mod n) for some t, 0 <= t < r.

gmpy2.**is_strong_selfridge_prp**(*n*, /) → bool

Return True if n is a strong Lucas probable prime with Selfidge parameters (p,q). The Selfridge parameters are chosen by finding the first element D in the sequence {5, -7, 9, -11, 13, ...} such that Jacobi(D,n) == -1. Then let p=1 and q = (1-D)/4. Then perform a strong Lucas probable prime test.

gmpy2.**lucasu**(*p*, *q*, *k*, /) → *mpz*

Return the k-th element of the Lucas U sequence defined by p,q. p*p - 4*q must not equal 0; k must be greater than or equal to 0.

gmpy2.**lucasu_mod**(*p*, *q*, *k*, *n*, /) → *mpz*

Return the k-th element of the Lucas U sequence defined by p,q (mod n). p*p - 4*q must not equal 0; k must be greater than or equal to 0; n must be greater than 0.

gmpy2.**lucasv**(*p*, *q*, *k*, /) → *mpz*

Return the k-th element of the Lucas V sequence defined by p,q. p*p - 4*q must not equal 0; k must be greater than or equal to 0.

gmpy2.**lucasv_mod**(*p*, *q*, *k*, *n*, /) → *mpz*

Return the k-th element of the Lucas V sequence defined by p,q (mod n). p*p - 4*q must not equal 0; k must be greater than or equal to 0; n must be greater than 0.

CHAPTER

SIX

MULTIPLE-PRECISION RATIONALS

gmpy2 provides a rational type *mpq*. It should be a replacement for Python's `Fraction` class.

```
>>> from gmpy2 import mpq
>>> mpq(1,7)
mpq(1,7)
>>> mpq(1,7) * 11
mpq(11,7)
>>> mpq(11,7)/13
mpq(11,91)
```

6.1 mpq type

class gmpy2.**mpq**(*n=0*, */*)

class gmpy2.**mpq**(*n*, *m*, */*)

class gmpy2.**mpq**(*s*, */*, *base=10*)

Return a rational number constructed from a non-complex number n exactly or from a pair of `Rational` values n and m or from a string s made up of digits in the given base. Every input, that is accepted by the `Fraction` type constructor is also accepted.

A string may be made up to two integers in the same base separated by a '/' character, both parsed the same as the *mpz* type constructor does. If base=10, any string that represents a finite value and is accepted by the `float` constructor is also accepted.

as_integer_ratio() → tuple[*mpz*, *mpz*]

Return a pair of integers, whose ratio is exactly equal to the original number. The ratio is in lowest terms and has a positive denominator.

conjugate() → *mpz*

Return the conjugate of x (which is just a new reference to x since x is not a complex number).

digits(*base=10*, */*) → str

Return a Python string representing x in the given base (2 to 62, default is 10). A leading '-' is present if x<0, but no leading '+' is present if x>=0.

from_decimal(*dec*, */*) → *mpq*

Converts a finite `decimal.Decimal` instance to a rational number, exactly.

from_float(*f*, */*) → *mpq*

Converts a finite float to a rational number, exactly.

denominator
: the denominator of a rational number in lowest terms

imag
: the imaginary part of a complex number

numerator
: the numerator of a rational number in lowest terms

real
: the real part of a complex number

6.2 mpq Functions

gmpy2.**qdiv**(*x*, *y=1*, */*) → *mpz* | *mpq*
: Return x/y as *mpz* if possible, or as *mpq* if x is not exactly divisible by y.

CHAPTER

SEVEN

CONTEXTS

A *context* type is used to control the behavior of *mpfr* and *mpc* arithmetic. In addition to controlling the precision, the rounding mode can be specified, minimum and maximum exponent values can be changed, various exceptions can be raised or ignored, gradual underflow can be enabled, and returning complex results can be enabled.

context() creates a new context with all options set to default. *set_context()* will set the active context. *get_context()* will return a reference to the active context. Note that contexts are mutable: modifying the reference returned by *get_context()* will modify the active context until a new context is enabled with *set_context()*. The *copy()* method of a context will return a copy of the context.

The following example just modifies the precision. The remaining options will be discussed later.

```
>>> import gmpy2
>>> from gmpy2 import mpfr
>>> gmpy2.set_context(gmpy2.context())
>>> gmpy2.get_context()
context(precision=53, real_prec=Default, imag_prec=Default,
        round=RoundToNearest, real_round=Default, imag_round=Default,
        emax=1073741823, emin=-1073741823,
        subnormalize=False,
        trap_underflow=False, underflow=False,
        trap_overflow=False, overflow=False,
        trap_inexact=False, inexact=False,
        trap_invalid=False, invalid=False,
        trap_erange=False, erange=False,
        trap_divzero=False, divzero=False,
        allow_complex=False,
        rational_division=False,
        allow_release_gil=False)
>>> gmpy2.sqrt(5)
mpfr('2.2360679774997898')
>>> gmpy2.get_context().precision=100
>>> gmpy2.sqrt(5)
mpfr('2.2360679774997896964091736687316',100)
>>> gmpy2.get_context().precision+=20
>>> gmpy2.sqrt(5)
mpfr('2.2360679774997896964091736687312762351',120)
>>> ctx=gmpy2.get_context()
>>> ctx.precision+=20
>>> gmpy2.sqrt(5)
mpfr('2.2360679774997896964091736687312762354406182',140)
>>> gmpy2.set_context(gmpy2.context())
```

(continues on next page)

(continued from previous page)

```
>>> gmpy2.sqrt(5)
mpfr('2.2360679774997898')
>>> ctx.precision+=20
>>> gmpy2.sqrt(5)
mpfr('2.2360679774997898')
>>> gmpy2.set_context(ctx)
>>> gmpy2.sqrt(5)
mpfr('2.2360679774997896964091736687312762354406183596116',160)
>>> gmpy2.set_context(gmpy2.context())
```

7.1 Context Type

class gmpy2.**context**

GMPY2 Context Object

abs(*x*, /) → *mpz* | *mpq* | *mpfr*

Return abs(x), the context is applied to the result.

acos(*x*, /) → *mpfr* | *mpc*

Return inverse cosine of x; result in radians.

acosh(*x*, /) → *mpfr* | *mpc*

Return inverse hyperbolic cosine of x.

add(*x*, *y*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return x + y.

agm(*x*, *y*, /) → *mpfr*

Return arithmetic-geometric mean of x and y.

ai(*x*, /) → *mpfr*

Return Airy function of x.

asin(*x*, /) → *mpfr* | *mpc*

Return inverse sine of x; result in radians.

asinh(*x*, /) → *mpfr* | *mpc*

Return inverse hyperbolic sine of x.

atan(*x*, /) → *mpfr* | *mpc*

Return inverse tangent of x; result in radians.

atan2(*y*, *x*, /) → *mpfr*

Return arc-tangent of (y/x); result in radians.

atanh(*x*, /) → *mpfr* | *mpc*

Return inverse hyperbolic tanget of x.

cbrt(*x*, /) → *mpfr*

Return the cube root of x.

ceil(*x*, /) → *mpfr*

Return an *mpfr* that is the smallest integer >= x.

check_range(*x*, /) → *mpfr*

Return a new *mpfr* with exponent that lies within the range of emin and emax specified by context.

clear_flags() → None

Clear all MPFR exception flags.

const_catalan() → *mpfr*

Return the catalan constant using the context's precision.

const_euler() → *mpfr*

Return the euler constant using the context's precision.

const_log2() → *mpfr*

Return the log2 constant using the context's precision.

const_pi() → *mpfr*

Return the constant pi using the context's precision.

copy() → *context*

Return a copy of a context.

cos(*x*, /) → *mpfr* | *mpc*

Return cosine of x; x in radians.

cosh(*x*, /) → *mpfr* | *mpc*

Return hyperbolic cosine of x.

cot(*x*, /) → *mpfr*

Return cotangent of x; x in radians.

coth(*x*, /) → *mpfr*

Return hyperbolic cotangent of x.

csc(*x*, /) → *mpfr*

Return cosecant of x; x in radians.

csch(*x*, /) → *mpfr*

Return hyperbolic cosecant of x.

degrees(*x*, /) → *mpfr*

Convert angle x from radians to degrees. Note: In rare cases the result may not be correctly rounded.

digamma(*x*, /) → *mpfr*

Return digamma of x.

div(*x*, *y*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return x / y; uses true division.

div_2exp(*x*, *n*, /) → *mpfr* | *mpc*

Return *mpfr* or *mpc* divided by 2**n.

divmod(*x*, *y*, /) → tuple[*mpz* | *mpfr*, *mpz* | *mpq* | *mpfr*]

Return divmod(x, y).

Note: overflow, underflow, and inexact exceptions are not supported for *mpfr* arguments.

eint(*x*, /) → *mpfr*

Return exponential integral of x.

erf(*x*, /) → *mpfr*

Return error function of x.

erfc(*x*, /) → *mpfr*

Return complementary error function of x.

exp(*x*, /) → *mpfr* | *mpc*

Return the exponential of x.

exp10(*x*, /) → *mpfr*

Return 10**x.

exp2(*x*, /) → *mpfr*

Return 2**x.

expm1(*x*, /) → *mpfr*

Return exp(x) - 1.

factorial(*n*, /) → *mpfr*

Return the floating-point approximation to the factorial of n.

See *fac()* to get the exact integer result.

floor(*x*, /) → *mpfr*

Return an *mpfr* that is the largest integer <= x.

floor_div(*x*, *y*, /) → *mpz* | *mpfr*

Return x // y; uses floor division.

fma(*x*, *y*, *z*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return correctly rounded result of (x * y) + z.

fmma(*x*, *y*, *z*, *t*, /) → *mpfr*

Return correctly rounded result of (x * y) + (z * t).

fmms(*x*, *y*, *z*, *t*, /) → *mpfr*

Return correctly rounded result of (x * y) - (z * t).

fmod(*x*, *y*, /) → *mpfr*

Return x - n*y where n is the integer quotient of x/y, rounded to 0.

fms(*x*, *y*, *z*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return correctly rounded result of (x * y) - z.

frac(*x*, /) → *mpfr*

Return fractional part of x.

frexp(*x*, /) → tuple[int, *mpfr*]

Return a `tuple` containing the exponent and mantissa of x.

fsum(*iterable*, /) → *mpfr*

Return an accurate sum of the values in the iterable.

gamma(*x*, /) → *mpfr*

Return gamma of x.

gamma_inc(*a*, *x*, /) → *mpfr*

Return (upper) incomplete gamma of a and x.

hypot(*x*, *y*, /) → *mpfr*

Return square root of (x**2 + y**2).

is_finite(*x*, /) → bool

Return `True` if x is an actual number (i.e. non NaN or Infinity). If x is an *mpc*, return `True` if both x.real and x.imag are finite.

is_infinite(*x*, /) → bool

Return `True` if x is +Infinity or -Infinity. If x is an *mpc*, return `True` if either x.real or x.imag is infinite. Otherwise return `False`.

is_integer(*x*, /) → bool

Return `True` if x is an integer; `False` otherwise.

is_nan(*x*, /) → bool

Return `True` if x is NaN (Not-A-Number) else `False`.

is_regular(*x*, /) → bool

Return `True` if x is not zero, NaN, or Infinity; `False` otherwise.

is_signed(*x*, /) → bool

Return `True` if the sign bit of x is set.

is_zero(*x*, /) → bool

Return `True` if x is equal to 0. If x is an *mpc*, return `True` if both x.real and x.imag are equal to 0.

j0(*x*, /) → *mpfr*

Return first kind Bessel function of order 0 of x.

j1(*x*, /) → *mpfr*

Return first kind Bessel function of order 1 of x.

jn(*x*, *n*, /) → *mpfr*

Return the first kind Bessel function of order n of x.

lgamma(*x*, /) → tuple[*mpfr*, int]

Return a `tuple` containing the logarithm of the absolute value of gamma(x) and the sign of gamma(x)

li2(*x*, /) → *mpfr*

Return real part of dilogarithm of x.

lngamma(*x*, /) → *mpfr*

Return natural logarithm of gamma(x).

log(*x*, /) → *mpfr* | *mpc*

Return the natural logarithm of x.

log10(*x*, /) → *mpfr* | *mpc*

Return the base-10 logarithm of x.

log1p(*x*, /) → *mpfr*

Return natural logarithm of (1+x).

log2(*x*, /) → *mpfr*

Return base-2 logarithm of x.

maxnum(*x*, *y*, /) → *mpfr*

Return the maximum number of x and y. If x and y are not *mpfr*, they are converted to *mpfr*. The result is rounded to match the specified context. If only one of x or y is a number, then that number is returned.

minnum(*x*, *y*, /) → *mpfr*

Return the minimum number of x and y. If x and y are not *mpfr*, they are converted to *mpfr*. The result is rounded to match the specified context. If only one of x or y is a number, then that number is returned.

minus(*x*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return -x. The context is applied to the result.

mod(*x*, *y*, /) → *mpz* | *mpq* | *mpfr*

Return mod(x, y). Note: overflow, underflow, and inexact exceptions are not supported for *mpfr* arguments.

modf(*x*, /) → tuple[*mpfr*, *mpfr*]

Return a `tuple` containing the integer and fractional portions of x.

mul(*x*, *y*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return x * y.

mul_2exp(*x*, *n*, /) → *mpfr* | *mpc*

Return *mpfr* or *mpc* multiplied by 2**n.

next_above(*x*, /) → *mpfr*

Return the next *mpfr* from x toward +Infinity.

next_below(*x*, /) → *mpfr*

Return the next *mpfr* from x toward -Infinity.

next_toward(*x*, *y*, /) → *mpfr*

Return the next *mpfr* from x in the direction of y. The result has the same precision as x.

norm(*x*, /) → *mpfr*

Return the norm of a complex x. The norm(x) is defined as x.real**2 + x.imag**2. abs(x) is the square root of norm(x).

phase(*x*, /) → *mpfr*

Return the phase angle, also known as argument, of a complex x.

plus(*x*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return +x, the context is applied to the result.

polar(*x*, /) → tuple[*mpfr*, *mpfr*]

Return the polar coordinate form of a complex x that is in rectangular form.

pow(*x*, *y*, /) → *mpz* | *mpq* | *mpfr* | *mpc*

Return x ** y.

proj(*x*, /) → *mpc*

Returns the projection of a complex x on to the Riemann sphere.

radians(*x*, /) → *mpfr*

Convert angle x from degrees to radians. Note: In rare cases the result may not be correctly rounded.

rec_sqrt(*x*, /) → *mpfr*

Return the reciprocal of the square root of x.

rect(*r*, *phi*, /) → *mpc*

Return the rectangular coordinate form of a complex number that is given in polar form.

reldiff(*x*, *y*, /) → *mpfr*

Return the relative difference between x and y. Result is equal to abs(x-y)/x.

remainder(*x*, *y*, /) → *mpfr*

Return x - n*y where n is the integer quotient of x/y, rounded to the nearest integer and ties rounded to even.

remquo(*x*, *y*, /) → tuple[*mpfr*, int]

Return a `tuple` containing the remainder(x,y) and the low bits of the quotient.

rint(*x*, /) → *mpfr*

Return x rounded to the nearest integer using the context rounding mode.

rint_ceil(*x*, /) → *mpfr*

Return x rounded to the nearest integer by first rounding to the next higher or equal integer and then, if needed, using the context rounding mode.

rint_floor(*x*, /) → *mpfr*

Return x rounded to the nearest integer by first rounding to the next lower or equal integer and then, if needed, using the context rounding mode.

rint_round(*x*, /) → *mpfr*

Return x rounded to the nearest integer by first rounding to the nearest integer (ties away from 0) and then, if needed, using the context rounding mode.

rint_trunc(*x*, /) → *mpfr*

Return x rounded to the nearest integer by first rounding towards zero and then, if needed, using the context rounding mode.

root(*x*, *n*, /) → *mpfr*

Return n-th root of x. The result always an *mpfr*. Note: not IEEE 754-2008 compliant; result differs when x = -0 and n is even. See *context.rootn()*.

root_of_unity(*n*, *k*, /) → *mpc*

Return the n-th root of mpc(1) raised to the k-th power..

rootn(*x*, *n*, /) → *mpfr*

Return n-th root of x. The result always an *mpfr*. Note: this is IEEE 754-2008 compliant version of *context.root()*.

round2(*x*, *n=0*, /) → *mpfr*

Return x rounded to n bits. Uses default precision if n is not specified. See *context.round_away()* to access the mpfr_round() function of the MPFR.

round_away(*x*, /) → *mpfr*

Return an *mpfr* that is x rounded to the nearest integer, with ties rounded away from 0.

sec(*x*, /) → *mpfr*

Return secant of x; x in radians.

sech(*x*, /) → *mpfr*

Return hyperbolic secant of x.

sin(*x*, /) → *mpfr* | *mpc*

Return sine of x; x in radians.

sin_cos(*x*, /) → tuple[*mpfr* | *mpc*, *mpfr* | *mpc*]
Return a tuple containing the sine and cosine of x; x in radians.

sinh(*x*, /) → *mpfr* | *mpc*
Return hyperbolic sine of x.

sinh_cosh(*x*, /) → tuple[*mpfr*, *mpfr*]
Return a `tuple` containing the hyperbolic sine and cosine of x.

sqrt(*x*, /) → *mpfr* | *mpc*
Return the square root of x.

square(*x*, /) → *mpz* | *mpq* | *mpfr* | *mpc*
Return x * x.

sub(*x*, *y*, /) → *mpz* | *mpq* | *mpfr* | *mpc*
Return x - y.

tan(*x*, /) → *mpfr* | *mpc*
Return tangent of x; x in radians.

tanh(*x*, /) → *mpfr* | *mpc*
Return hyperbolic tangent of x.

trunc(*x*, /) → *mpfr*
Return an *mpfr* that is x truncated towards 0. Same as x.floor() if x>=0 or x.ceil() if x<0.

y0(*x*, /) → *mpfr*
Return second kind Bessel function of order 0 of x.

y1(*x*, /) → *mpfr*
Return second kind Bessel function of order 1 of x.

yn(*x*, *n*, /) → *mpfr*
Return the second kind Bessel function of order n of x.

zeta(*x*, /) → *mpfr*
Return Riemann zeta of x.

allow_complex
This attribute controls whether or not an *mpc* result can be returned if an *mpfr* result would normally not be possible.

allow_release_gil
If set to `True`, many *mpz* and *mpq* computations will release the GIL.

This is considered an experimental feature.

divzero
This flag is not user controllable. It is automatically set if a division by zero occurred and NaN result was returned.

emax
This attribute controls the maximum allowed exponent of an *mpfr* result. The maximum exponent is platform dependent and can be retrieved with *get_emax_max()*.

emin
This attribute controls the minimum allowed exponent of an *mpfr* result. The minimum exponent is platform dependent and can be retrieved with *get_emin_min()*.

erange

This flag is not user controllable. It is automatically set if an erange error occurred.

imag_prec

This attribute controls the precision of the imaginary part of an *mpc* result. If the value is Default, then the value of *real_prec* is used.

imag_round

This attribute controls the rounding mode for the imaginary part of an *mpc* result. If the value is Default, then the value of the *real_round* attribute is used. Note: RoundAwayZero is not a valid rounding mode for *mpc*.

inexact

This flag is not user controllable. It is automatically set if an inexact result is returned.

invalid

This flag is not user controllable. It is automatically set if an invalid (Not-A-Number) result is returned.

overflow

This flag is not user controllable. It is automatically set if a result overflowed to +/-Infinity and *trap_overflow* is `False`.

precision

This attribute controls the precision of an *mpfr* result. The precision is specified in bits, not decimal digits. The maximum precision that can be specified is platform dependent and can be retrieved with *get_max_precision()*.

Note: Specifying a value for precision that is too close to the maximum precision will cause the MPFR library to fail.

rational_division

If set to `True`, *mpz* / *mpz* will return an *mpq* instead of an *mpfr*.

real_prec

This attribute controls the precision of the real part of an *mpc* result. If the value is Default, then the value of the *precision* attribute is used.

real_round

This attribute controls the rounding mode for the real part of an *mpc* result. If the value is Default, then the value of the round attribute is used. Note: RoundAwayZero is not a valid rounding mode for *mpc*.

round

There are five rounding modes available to *mpfr* type:

- RoundAwayZero - The result is rounded away from 0.0.
- RoundDown - The result is rounded towards -Infinity.
- RoundToNearest - Round to the nearest value; ties are rounded to an even value.
- RoundToZero - The result is rounded towards 0.0.
- RoundUp - The result is rounded towards +Infinity.

subnormalize

The usual IEEE-754 floating point representation supports gradual underflow when the minimum exponent is reached. The MFPR library does not enable gradual underflow by default but it can be enabled to precisely mimic the results of IEEE-754 floating point operations.

trap_divzero

This attribute controls whether or not a *DivisionByZeroError* exception is raised if division by 0 occurs. The *DivisionByZeroError* is a sub-class of Python's ZeroDivisionError.

trap_erange

This attribute controls whether or not a *RangeError* exception is raised when certain operations are performed on NaN and/or Infinity values. Setting *trap_erange* to True can be used to raise an exception if comparisons are attempted with a NaN.

trap_inexact

This attribute controls whether or not an *InexactResultError* exception is raised if an inexact result is returned. To check if the result is greater or less than the exact result, check the rc attribute of the *mpfr* result.

trap_invalid

This attribute controls whether or not an *InvalidOperationError* exception is raised if a numerical result is not defined. A special NaN (Not-A-Number) value will be returned if an exception is not raised. The *InvalidOperationError* is a sub-class of Python's ValueError.

For example, gmpy2.sqrt(-2) will normally return mpfr('nan'). However, if *allow_complex* is set to True, then an *mpc* result will be returned.

trap_overflow

If set to False, a result that is larger than the largest possible *mpfr* given the current exponent range will be replaced by +/-Infinity. If set to True, an *OverflowResultError* exception is raised.

trap_underflow

If set to False, a result that is smaller than the smallest possible *mpfr* given the current exponent range will be replaced by +/-0.0. If set to True, an *UnderflowResultError* exception is raised.

underflow

This flag is not user controllable. It is automatically set if a result underflowed to +/-0.0 and *trap_underflow* is False.

7.2 Context Functions

gmpy2.**get_context**() → *context*

Return a reference to the current context.

gmpy2.**ieee**(*size*, */*, *subnormalize=True*) → *context*

Return a new context corresponding to a standard IEEE floating point format. The supported sizes are 16, 32, 64, 128, and multiples of 32 greater than 128.

gmpy2.**local_context**(***kwargs*) → *context*

gmpy2.**local_context**(*context*, */*, ***kwargs*) → *context*

Create a context manager object that will restore the current context when the 'with ...' block terminates. The temporary context for the 'with ...' block is based on the current context if no context is specified. Keyword arguments are supported and will modify the temporary new context.

gmpy2.**set_context**(*context*, */*) → None

Activate a context object controlling gmpy2 arithmetic.

7.3 Contexts and the with statement

Contexts can also be used in conjunction with Python's `with` statement to temporarily change the context settings for a block of code and then restore the original settings when the block of code exits.

local_context first save the current context and then creates a new context based on a context passed as the first argument, or the current context if no context is passed. The new context is modified if any optional keyword arguments are given. The original active context is restored when the block completes.

In the following example, the current context is saved by *local_context* and then the block begins with a copy of the default context and the precision set to 100. When the block is finished, the original context is restored.

```
>>> print(gmpy2.sqrt(2))
1.4142135623730951
>>> with gmpy2.local_context(gmpy2.context(), precision=100) as ctx:
...   print(gmpy2.sqrt(2))
...   ctx.precision += 100
...   print(gmpy2.sqrt(2))
...
1.4142135623730950488016887242092
1.4142135623730950488016887242096980785696718753769480731766796
>>> print(gmpy2.sqrt(2))
1.4142135623730951
```

Contexts that implement the standard *single*, *double*, and *quadruple* precision floating point types can be created using *ieee*.

CHAPTER

EIGHT

MULTIPLE-PRECISION REALS

The *mpfr* type is based on the MPFR library. The new *mpfr* type supports correct rounding, selectable rounding modes, and many trigonometric, exponential, and special functions. A *context manager* is used to control precision, rounding modes, and the behavior of exceptions.

The default precision of an *mpfr* is 53 bits - the same precision as Python's `float` type. If the precision is changed, then `mpfr(float('1.2'))` differs from `mpfr('1.2')`. To take advantage of the higher precision provided by the *mpfr* type, always pass constants as strings.

```
>>> import gmpy2
>>> from gmpy2 import mpfr
>>> gmpy2.set_context(gmpy2.context())
>>> mpfr('1.2')
mpfr('1.2')
>>> mpfr(float('1.2'))
mpfr('1.2')
>>> gmpy2.get_context().precision=100
>>> mpfr('1.2')
mpfr('1.2000000000000000000000000000006',100)
>>> mpfr(float('1.2'))
mpfr('1.1999999999999999955591079014994',100)
```

The *mpfr* type supports the *__format__()* special method to allow custom output formatting.

```
>>> from gmpy2 import mpfr
>>> a=mpfr("1.23456")
>>> "{0:15.3f}".format(a)
'          1.235'
>>> "{0:15.3Uf}".format(a)
'          1.235'
>>> "{0:15.3Df}".format(a)
'          1.234'
>>> "{0:.3Df}".format(a)
'1.234'
>>> "{0:+.3Df}".format(a)
'+1.234'
```

8.1 mpfr Type

class gmpy2.**mpfr**(*n=0*, */*, *precision=0*)

class gmpy2.**mpfr**(*n*, */*, *precision*, *context*)

class gmpy2.**mpfr**(*s*, */*, *precision=0*, *base=0*)

class gmpy2.**mpfr**(*s*, */*, *precision*, *base*, *context*)

Return a floating-point number after converting a numeric value n or a string s made of digits in the given base.

A string can be with fraction-part (with a period as a separator) and/or exponent-part (with an exponent marker 'e' for base<=10, else '@'), where digits are parsed the same as the *mpz* type constructor does and both the whole number and exponent-part optionally can be preceded by '+' or '-'.

If a precision greater than or equal to 2 is specified, then it is used. A precision of 0 (the default) implies the precision of either the specified context or the current context is used. A precision of 1 minimizes the loss of precision by following these rules:

1) If n is a radix-2 floating point number, then the full precision of n is retained.

2) If n is an integer, then the precision is the bit length of the integer.

__format__(*fmt*) → str

Return a Python string by formatting 'x' using the format string 'fmt'. A valid format string consists of:

optional alignment code:

'<' -> left shifted in field '>' -> right shifted in field '^' -> centered in field

optional leading sign code

'+' -> always display leading sign '-' -> only display minus for negative values ' ' -> minus for negative values, space for positive values

optional width.precision

optional rounding mode:

'U' -> round toward plus Infinity 'D' -> round toward minus Infinity 'Y' -> round away from zero 'Z' -> round toward zero 'N' -> round to nearest

optional conversion code:

'a','A' -> hex format 'b' -> binary format 'e','E' -> scientific format 'f','F' -> fixed point format 'g','G' -> fixed or float format

The default format is '.6f'.

as_integer_ratio() → tuple[*mpz*, *mpz*]

Return the exact rational equivalent of an *mpfr*. Value is a `tuple` for compatibility with Python's `float.as_integer_ratio`.

as_mantissa_exp() → tuple[*mpz*, *mpz*]

Return the mantissa and exponent of an *mpfr*.

as_simple_fraction(*precision=0*) → *mpq*

Return a simple rational approximation to x. The result will be accurate to 'precision' bits. If 'precision' is 0, the precision of 'x' will be used.

conjugate() → *mpz*

Return the conjugate of x (which is just a new reference to x since x is not a complex number).

digits(*base=10*, *prec=0*, */*) → tuple[str, int, int]

Returns up to 'prec' digits in the given base. If 'prec' is 0, as many digits that are available are returned. No more digits than available given x's precision are returned. 'base' must be between 2 and 62, inclusive. The result is a three element `tuple` containing the mantissa, the exponent, and the number of bits of precision.

is_finite() → bool

Return `True` if x is an actual number (i.e. non NaN or Infinity). If x is an *mpc*, return `True` if both x.real and x.imag are finite.

is_infinite() → bool

Return `True` if x is +Infinity or -Infinity. If x is an *mpc*, return `True` if either x.real or x.imag is infinite. Otherwise return `False`.

is_integer() → bool

Return `True` if x is an integer; `False` otherwise.

is_nan() → bool

Return `True` if x is NaN (Not-A-Number) else `False`.

is_regular() → bool

Return `True` if x is not zero, NaN, or Infinity; `False` otherwise.

is_signed() → bool

Return `True` if the sign bit of x is set.

is_zero() → bool

Return `True` if x is equal to 0. If x is an *mpc*, return `True` if both x.real and x.imag are equal to 0.

imag

imaginary component

precision

precision in bits

rc

return code

real

real component

8.2 mpfr Functions

gmpy2.**agm**(*x*, *y*, */*) → *mpfr*

Return arithmetic-geometric mean of x and y.

gmpy2.**ai**(*x*, */*) → *mpfr*

Return Airy function of x.

gmpy2.**atan2**(*y*, *x*, */*) → *mpfr*

Return arc-tangent of (y/x); result in radians.

gmpy2.**cbrt**(*x*, */*) → *mpfr*

Return the cube root of x.

gmpy2.**ceil**(*x*, /) → *mpfr*

Return an 'mpfr' that is the smallest integer >= x.

gmpy2.**check_range**(*x*, /) → *mpfr*

Return a new *mpfr* with exponent that lies within the current range of emin and emax.

gmpy2.**cmp**(*x*, *y*, /) → int

Return -1 if x < y; 0 if x = y; or 1 if x > y. Both x and y must be integer, rational or real. Note: 0 is returned (and exception flag set) if either argument is NaN.

gmpy2.**const_catalan**(*precision=0*) → *mpfr*

Return the catalan constant using the specified precision. If no precision is specified, the default precision is used.

gmpy2.**const_euler**(*precision=0*) → *mpfr*

Return the euler constant using the specified precision. If no precision is specified, the default precision is used.

gmpy2.**const_log2**(*precision=0*) → *mpfr*

Return the log2 constant using the specified precision. If no precision is specified, the default precision is used.

gmpy2.**const_pi**(*precision=0*) → *mpfr*

Return the constant pi using the specified precision. If no precision is specified, the default precision is used.

gmpy2.**cot**(*x*, /) → *mpfr*

Return cotangent of x; x in radians.

gmpy2.**coth**(*x*, /) → *mpfr*

Return hyperbolic cotangent of x.

gmpy2.**csc**(*x*, /) → *mpfr*

Return cosecant of x; x in radians.

gmpy2.**csch**(*x*, /) → *mpfr*

Return hyperbolic cosecant of x.

gmpy2.**degrees**(*x*, /) → *mpfr*

Convert angle x from radians to degrees. Note: In rare cases the result may not be correctly rounded.

gmpy2.**digamma**(*x*, /) → *mpfr*

Return digamma of x.

gmpy2.**eint**(*x*, /) → *mpfr*

Return exponential integral of x.

gmpy2.**erf**(*x*, /) → *mpfr*

Return error function of x.

gmpy2.**erfc**(*x*, /) → *mpfr*

Return complementary error function of x.

gmpy2.**exp10**(*x*, /) → *mpfr*

Return 10**x.

gmpy2.**exp2**(*x*, /) → *mpfr*

Return 2**x.

gmpy2.**expm1**(*x*, /) → *mpfr*

Return exp(x) - 1.

gmpy2.**factorial**(*n*, /) → *mpfr*

Return the floating-point approximation to the factorial of n.

See *fac()* to get the exact integer result.

gmpy2.**floor**(*x*, /) → *mpfr*

Return an *mpfr* that is the largest integer <= x.

gmpy2.**fmma**(*x*, *y*, *z*, *t*, /) → *mpfr*

Return correctly rounded result of (x * y) + (z + t).

gmpy2.**fmms**(*x*, *y*, *z*, *t*, /) → *mpfr*

Return correctly rounded result of (x * y) - (z + t).

gmpy2.**fmod**(*x*, *y*, /) → *mpfr*

Return x - n*y where n is the integer quotient of x/y, rounded to 0.

gmpy2.**frac**(*x*, /) → *mpfr*

Return fractional part of x.

gmpy2.**frexp**(*x*, /) → tuple[int, *mpfr*]

Return a `tuple` containing the exponent and mantissa of x.

gmpy2.**fsum**(*iterable*, /) → *mpfr*

Return an accurate sum of the values in the iterable.

gmpy2.**gamma**(*x*, /) → *mpfr*

Return gamma of x.

gmpy2.**gamma_inc**(*a*, *x*, /) → *mpfr*

Return (upper) incomplete gamma of a and x.

gmpy2.**get_exp**(*x*, /) → int

Return the exponent of x. Returns 0 for NaN or Infinity and sets the *context.erange* flag of the current context and will raise an exception if *context.trap_erange* is set.

gmpy2.**hypot**(*x*, *y*, /) → *mpfr*

Return square root of (x**2 + y**2).

gmpy2.**inf**(*n*, /) → *mpfr*

Return an *mpfr* initialized to Infinity with the same sign as n. If n is not given, +Infinity is returned.

gmpy2.**is_finite**(*x*, /) → bool

Return `True` if x is an actual number (i.e. non NaN or Infinity). If x is an *mpc*, return `True` if both x.real and x.imag are finite.

gmpy2.**is_infinite**(*x*, /) → bool

Return `True` if x is +Infinity or -Infinity. If x is an *mpc*, return `True` if either x.real or x.imag is infinite. Otherwise return `False`.

gmpy2.**is_regular**(*x*, /) → bool

Return `True` if x is not zero, NaN, or Infinity; `False` otherwise.

gmpy2.**is_signed**(*x*, /) → bool

Return `True` if the sign bit of x is set.

gmpy2.**is_unordered**(*x*, *y*, /) → bool

Return `True` if either x and/or y is NaN.

gmpy2.**j0**(*x*, /) → *mpfr*

Return first kind Bessel function of order 0 of x.

gmpy2.**j1**(*x*, /) → *mpfr*

Return first kind Bessel function of order 1 of x.

gmpy2.**jn**(*x*, *n*, /) → *mpfr*

Return the first kind Bessel function of order n of x.

gmpy2.**lgamma**(*x*, /) → tuple[*mpfr*, int]

Return a `tuple` containing the logarithm of the absolute value of gamma(x) and the sign of gamma(x)

gmpy2.**li2**(*x*, /) → *mpfr*

Return real part of dilogarithm of x.

gmpy2.**lngamma**(*x*, /) → *mpfr*

Return natural logarithm of gamma(x).

gmpy2.**log1p**(*x*, /) → *mpfr*

Return natural logarithm of (1+x).

gmpy2.**log2**(*x*, /) → *mpfr*

Return base-2 logarithm of x.

gmpy2.**maxnum**(*x*, *y*, /) → *mpfr*

Return the maximum number of x and y. If x and y are not *mpfr*, they are converted to *mpfr*. The result is rounded to match the current context. If only one of x or y is a number, then that number is returned.

gmpy2.**minnum**(*x*, *y*, /) → *mpfr*

Return the minimum number of x and y. If x and y are not *mpfr*, they are converted to *mpfr*. The result is rounded to match the current context. If only one of x or y is a number, then that number is returned.

gmpy2.**modf**(*x*, /) → tuple[*mpfr*, *mpfr*]

Return a `tuple` containing the integer and fractional portions of x.

gmpy2.**mpfr_from_old_binary**(*string*, /) → *mpfr*

Return an *mpfr* from a GMPY 1.x binary mpf format.

gmpy2.**mpfr_grandom**(*random_state*, /) → tuple[*mpfr*, *mpfr*]

Return two random numbers with gaussian distribution.

gmpy2.**mpfr_random**(*random_state*, /) → *mpfr*

Return uniformly distributed number between [0,1].

gmpy2.**nan**() → *mpfr*

Return an *mpfr* initialized to NaN (Not-A-Number).

gmpy2.**next_above**(*x*, /) → *mpfr*

Return the next *mpfr* from x toward +Infinity.

gmpy2.**next_below**(*x*, /) → *mpfr*

Return the next *mpfr* from x toward -Infinity.

gmpy2.**radians**(*x*, /) → *mpfr*

Convert angle x from degrees to radians. Note: In rare cases the result may not be correctly rounded.

gmpy2.**rec_sqrt**(*x*, /) → *mpfr*

Return the reciprocal of the square root of x.

gmpy2.**reldiff**(*x*, *y*, /) → *mpfr*

Return the relative difference between x and y. Result is equal to abs(x-y)/x.

gmpy2.**remainder**(*x*, *y*, /) → *mpfr*

Return x - n*y where n is the integer quotient of x/y, rounded to the nearest integer and ties rounded to even.

gmpy2.**remquo**(*x*, *y*, /) → tuple[*mpfr*, int]

Return a `tuple` containing the remainder(x,y) and the low bits of the quotient.

gmpy2.**rint**(*x*, /) → *mpfr*

Return x rounded to the nearest integer using the current rounding mode.

gmpy2.**rint_ceil**(*x*, /) → *mpfr*

Return x rounded to the nearest integer by first rounding to the next higher or equal integer and then, if needed, using the current rounding mode.

gmpy2.**rint_floor**(*x*, /) → *mpfr*

Return x rounded to the nearest integer by first rounding to the next lower or equal integer and then, if needed, using the current rounding mode.

gmpy2.**rint_round**(*x*, /) → *mpfr*

Return x rounded to the nearest integer by first rounding to the nearest integer (ties away from 0) and then, if needed, using the current rounding mode.

gmpy2.**rint_trunc**(*x*, /) → *mpfr*

Return x rounded to the nearest integer by first rounding towards zero and then, if needed, using the current rounding mode.

gmpy2.**root**(*x*, *n*, /) → *mpfr*

Return n-th root of x. The result always an *mpfr*. Note: not IEEE 754-2008 compliant; result differs when x = -0 and n is even. See *rootn()*.

gmpy2.**rootn**(*x*, *n*, /) → *mpfr*

Return n-th root of x. The result always an *mpfr*. Note: this is IEEE 754-2008 compliant version of *root()*.

gmpy2.**round2**(*x*, *n=0*, /) → *mpfr*

Return x rounded to n bits. Uses default precision if n is not specified. See *round_away()* to access the mpfr_round() function of the MPFR.

gmpy2.**round_away**(*x*, /) → *mpfr*

Return an *mpfr* that is x rounded to the nearest integer, with ties rounded away from 0.

gmpy2.**sec**(*x*, /) → *mpfr*

Return secant of x; x in radians.

gmpy2.**sech**(*x*, /) → *mpfr*

Return hyperbolic secant of x.

gmpy2.**set_exp**(*x*, *n*, /) → *mpfr*

Set the exponent of x to n. If n is outside the range of valid exponents, *set_exp()* will set the *context.erange* flag of the current context and either return the original value or raise an exception if *context.trap_erange* is set.

gmpy2.**set_sign**(*x*, *s*, /) → *mpfr*

If s is `True`, then return x with the sign bit set.

gmpy2.**sign**(*x*, /) → int

Return -1 if x < 0, 0 if x == 0, or +1 if x >0.

gmpy2.**sinh_cosh**(*x*, /) → tuple[*mpfr*, *mpfr*]

Return a `tuple` containing the hyperbolic sine and cosine of x.

gmpy2.**trunc**(*x*, /) → *mpfr*

Return an *mpfr* that is x truncated towards 0. Same as x.floor() if x>=0 or x.ceil() if x<0.

gmpy2.**y0**(*x*, /) → *mpfr*

Return second kind Bessel function of order 0 of x.

gmpy2.**y1**(*x*, /) → *mpfr*

Return second kind Bessel function of order 1 of x.

gmpy2.**yn**(*x*, *n*, /) → *mpfr*

Return the second kind Bessel function of order n of x.

gmpy2.**zero**(*n*, /) → *mpfr*

Return an *mpfr* initialized to 0.0 with the same sign as n. If n is not given, +0.0 is returned.

gmpy2.**zeta**(*x*, /) → *mpfr*

Return Riemann zeta of x.

gmpy2.**get_max_precision**() → int

Return the maximum bits of precision that can be used for calculations. Note: to allow extra precision for intermediate calculations, avoid setting precision close the maximum precision.

gmpy2.**get_emax_max**() → int

Return the maximum possible exponent that can be set for *mpfr*.

gmpy2.**get_emin_min**() → int

Return the minimum possible exponent that can be set for *mpfr*.

gmpy2.**copy_sign**(*x*, *y*, /) → *mpfr*

Return an *mpfr* composed of x with the sign of y.

gmpy2.**can_round**(*b*, *err*, *rnd1*, *rnd2*, *prec*, /) → bool

Let b be an approximation to an unknown number x that is rounded according to rnd1. Assume the b has an error at most two to the power of E(b)-err where E(b) is the exponent of b. Then return `True` if x can be rounded correctly to prec bits with rounding mode rnd2.

gmpy2.**free_cache**() → None

Free the internal cache of constants maintained by MPFR.

CHAPTER

NINE

MULTIPLE-PRECISION COMPLEX

gmpy2 adds a multiple-precision complex type called *mpc* that is based on the MPC library. The context manager settings for *mpfr* arithmetic are applied to *mpc* arithmetic by default. It is possible to specify different precision and rounding modes for both the real and imaginary components of an *mpc*.

```
>>> import gmpy2
>>> from gmpy2 import mpc
>>> gmpy2.set_context(gmpy2.context())
>>> gmpy2.sqrt(mpc("1+2j"))
mpc('1.272019649514069+0.78615137775742328j')
>>> gmpy2.set_context(gmpy2.context(real_prec=60,imag_prec=70))
>>> gmpy2.get_context()
context(precision=53, real_prec=60, imag_prec=70,
        round=RoundToNearest, real_round=Default, imag_round=Default,
        emax=1073741823, emin=-1073741823,
        subnormalize=False,
        trap_underflow=False, underflow=False,
        trap_overflow=False, overflow=False,
        trap_inexact=False, inexact=False,
        trap_invalid=False, invalid=False,
        trap_erange=False, erange=False,
        trap_divzero=False, divzero=False,
        allow_complex=False,
        rational_division=False,
        allow_release_gil=False)
>>> gmpy2.sqrt(mpc("1+2j"))
mpc('1.272019649514068965+0.78615137775742328606947j',(60,70))
>>> gmpy2.set_context(gmpy2.context())
```

Exceptions are normally raised in Python when the result of a real operation is not defined over the reals; for example, `sqrt(-4)` will raise an exception. The default context in gmpy2 implements the same behavior but by setting allow_complex to True, complex results will be returned.

```
>>> import gmpy2
>>> from gmpy2 import mpc
>>> gmpy2.sqrt(-4)
mpfr('nan')
>>> gmpy2.get_context().allow_complex=True
>>> gmpy2.sqrt(-4)
mpc('0.0+2.0j')
```

The *mpc* type supports the *__format__()* special method to allow custom output formatting.

```
>>> import gmpy2
>>> from gmpy2 import mpc
>>> a=gmpy2.sqrt(mpc("1+2j"))
>>> a
mpc('1.272019649514069+0.78615137775742328j')
>>> "{0:.4.4Mf}".format(a)
'(1.2720 0.7862)'
>>> "{0:.4.4f}".format(a)
'1.2720+0.7862j'
>>> "{0:^20.4.4U}".format(a)
'   1.2721+0.7862j   '
>>> "{0:^20.4.4D}".format(a)
'   1.2720+0.7861j   '
```

9.1 mpc Type

class gmpy2.**mpc**(*c=0*, */*, *precision=0*)

class gmpy2.**mpc**(*c=0*, */*, *precision*, *context*)

class gmpy2.**mpc**(*real*, */*, *imag=0*, *precision=0*)

class gmpy2.**mpc**(*real*, */*, *imag*, *precision*, *context*)

class gmpy2.**mpc**(*s*, */*, *precision=0*, *base=10*)

class gmpy2.**mpc**(*s*, */*, *precision*, *base*, *context*)

Return a complex floating-point number constructed from a numeric value c or from a pair of two non-complex numbers real and imag or from a string s made of digits in the given base.

A string can be possibly with real-part and/or imaginary-part (that have 'j' as a suffix), separated by '+' and parsed the same as the *mpfr* constructor does (but the base must be up to 36).

The precision can be specified by either a single number that is used for both the real and imaginary components, or as a pair of different precisions for the real and imaginary components. For every component, the meaning of its precision value is the same as in the *mpfr* type constructor.

__format__(*fmt*) → str

Return a Python string by formatting 'x' using the format string 'fmt'. A valid format string consists of:

optional alignment code:

'<' -> left shifted in field '>' -> right shifted in field '^' -> centered in field

optional leading sign code

'+' -> always display leading sign '-' -> only display minus for negative values ' ' -> minus for negative values, space for positive values

optional width.real_precision.imag_precision

optional rounding mode:

'U' -> round toward plus infinity 'D' -> round toward minus infinity 'Z' -> round toward zero 'N' -> round to nearest

optional output style:

'P' -> Python style, 1+2j, (default) 'M' -> MPC style, (1 2)

optional conversion code:

'a','A' -> hex format 'b' -> binary format 'e','E' -> scientific format 'f','F' -> fixed point format 'g','G' -> fixed or scientific format

The default format is 'f'.

conjugate() → *mpc*

Returns the conjugate of x.

digits(*base=10*, *prec=0*, /) → tuple[tuple[str, int, int], tuple[str, int, int]]

Returns up to 'prec' digits in the given base. If 'prec' is 0, as many digits that are available given c's precision are returned. 'base' must be between 2 and 62. The result consists of 2 three-element tuples that contain the mantissa, exponent, and number of bits of precision of the real and imaginary components.

is_finite() → bool

Return `True` if x is an actual number (i.e. non NaN or Infinity). If x is an *mpc*, return `True` if both x.real and x.imag are finite.

is_infinite() → bool

Return `True` if x is +Infinity or -Infinity. If x is an *mpc*, return `True` if either x.real or x.imag is infinite. Otherwise return `False`.

is_nan() → bool

Return `True` if x is NaN (Not-A-Number) else `False`.

is_zero() → bool

Return `True` if x is equal to 0. If x is an *mpc*, return `True` if both x.real and x.imag are equal to 0.

imag

imaginary component

precision

precision in bits

rc

return code

real

real component

9.2 mpc Functions

gmpy2.**acos**(*x*, /) → *mpfr* | *mpc*

Return inverse cosine of x; result in radians.

gmpy2.**acosh**(*x*, /) → *mpfr* | *mpc*

Return inverse hyperbolic cosine of x.

gmpy2.**asin**(*x*, /) → *mpfr* | *mpc*

Return inverse sine of x; result in radians.

gmpy2.**asinh**(*x*, /) → *mpfr* | *mpc*

Return inverse hyperbolic sine of x.

gmpy2.**atan**(*x*, /) → *mpfr* | *mpc*

Return inverse tangent of x; result in radians.

gmpy2.**atanh**(*x*, /) → *mpfr* | *mpc*

Return inverse hyperbolic tangent of x.

gmpy2.**cos**(*x*, /) → *mpfr* | *mpc*

Return cosine of x; x in radians.

gmpy2.**cosh**(*x*, /) → *mpfr* | *mpc*

Return hyperbolic cosine of x.

gmpy2.**div_2exp**(*x*, *n*, /) → *mpfr* | *mpc*

Return x divided by 2**n.

gmpy2.**exp**(*x*, /) → *mpfr* | *mpc*

Return the exponential of x.

gmpy2.**is_nan**(*x*, /) → bool

Return `True` if x is NaN (Not-A-Number) else `False`.

gmpy2.**is_zero**(*x*, /) → bool

Return `True` if x is equal to 0. If x is an *mpc*, return `True` if both x.real and x.imag are equal to 0.

gmpy2.**log**(*x*, /) → *mpfr* | *mpc*

Return the natural logarithm of x.

gmpy2.**log10**(*x*, /) → *mpfr* | *mpc*

Return the base-10 logarithm of x.

gmpy2.**mpc_random**(*random_state*, /) → *mpc*

Return uniformly distributed number in the unit square [0,1]x[0,1].

gmpy2.**mul_2exp**(*x*, *n*, /) → *mpfr* | *mpc*

Return x multiplied by 2**n.

gmpy2.**norm**(*x*, /) → *mpfr*

Return the norm of a complex x. The norm(x) is defined as x.real**2 + x.imag**2. abs(x) is the square root of norm(x).

gmpy2.**phase**(*x*, /) → *mpfr*

Return the phase angle, also known as argument, of a complex x.

gmpy2.**polar**(*x*, /) → tuple[*mpfr*, *mpfr*]

Return the polar coordinate form of a complex x that is in rectangular form.

gmpy2.**proj**(*x*, /) → *mpc*

Returns the projection of a complex x on to the Riemann sphere.

gmpy2.**rect**(*r*, *phi*, /) → *mpc*

Return the rectangular coordinate form of a complex number that is given in polar form.

gmpy2.**root_of_unity**(*n*, *k*, /) → *mpc*

Return the n-th root of mpc(1) raised to the k-th power..

gmpy2.**sin**(*x*, /) → *mpfr* | *mpc*

Return sine of x; x in radians.

gmpy2.**sin_cos**(*x*, /) → tuple[*mpfr* | *mpc*, *mpfr* | *mpc*]

Return a tuple containing the sine and cosine of x; x in radians.

gmpy2.**sinh**(*x*, /) → *mpfr* | *mpc*

Return hyperbolic sine of x.

gmpy2.**sqrt**(*x*, /) → *mpfr* | *mpc*

Return the square root of x.

gmpy2.**tan**(*x*, /) → *mpfr* | *mpc*

Return tangent of x; x in radians.

gmpy2.**tanh**(*x*, /) → *mpfr* | *mpc*

Return hyperbolic tangent of x.

CHAPTER

TEN

CYTHON USAGE

The gmpy2 module provides a C-API that can be conveniently used from Cython. All types and functions are declared in the header gmpy2.pxd that is installed automatically in your Python path together with the library.

10.1 Initialization

In order to use the C-API you need to make one call to the function **void import_gmpy2(void)**.

10.2 Types

The types *mpz*, *mpq*, *mpfr* and *mpc* are declared as extension types in gmpy2.pxd. They correspond respectively to the C structures **MPZ_Object**, **MPQ_Object**, **MPFR_Object** and **MPC_Object**.

Fast type checking can be done with the following C functions

bint MPZ_Check(object)
: equivalent to **isinstance(obj, mpz)**

bint MPQ_Check(object)
: equivalent to **isinstance(obj, mpq)**

bint MPFR_Check(object)
: equivalent to **isinstance(obj, mpfr)**

bint MPC_Check(object)
: equivalent to **isinstance(obj, mpc)**

10.3 Object creation

To create a new gmpy2 types there are four basic functions

mpz GMPy_MPZ_New(void * ctx)
: create a new mpz object from a given context ctx

mpq GMPy_MPQ_New(void * ctx)
: create a new mpq object from a given context ctx

mpfr MPFR_New(void * ctx, mpfr_prec_t prec)
: create a new mpfr object with given context ctx and precision prec

mpc MPC_New(void * ctx, mpfr_prec_t rprec, mpfr_prec_t iprec)
create a new mpc object with given context ctx, precisions rprec and iprec of respectively real and imaginary parts

The context can be set to **NULL** and controls the default behavior (e.g. precision).

The gmpy2.pxd header also provides convenience macro to wrap a (copy of) a mpz_t, mpq_t, mpfr_t or a mpc_t object into the corresponding gmpy2 type.

mpz GMPy_MPZ_From_mpz(mpz_srcptr z)
return a new mpz object with a given mpz_t value z

mpq GMPy_MPQ_From_mpq(mpq_srcptr q)
return a new mpq object from a given mpq_t value q

mpq GMPy_MPQ_From_mpz(mpz_srcptr num, mpz_srcptr den)
return a new mpq object with a given mpz_t numerator num and mpz_t denominator den

mpfr GMPy_MPFR_From_mpfr(mpfr_srcptr x)
return a new mpfr object with a given mpfr_t value x

mpc GMPy_MPC_From_mpc(mpc_srcptr c)
return a new mpc object with a given mpc_t value c

mpc GMPy_MPC_From_mpfr(mpfr_srcptr re, mpfr_srcptr im)
return a new mpc object with a given mpfr_t real part re and mpfr_t imaginary part im

10.4 Access to the underlying C type

Each of the gmpy2 objects has a field corresponding to the underlying C type. The following functions give access to this field

mpz_t MPZ(mpz)

mpq_t MPQ(mpq)

mpfr_t MPFR(mpfr)

mpc_t MPC(mpc)

10.5 Compilation

The header gmpy2.pxd as well as the C header gmpy2.h from which it depends are installed in the Python path. In order to make Cython and the C compiler aware of the existence of these files, the Python path should be part of the include directories.

Recall that **import_gmpy2()** needs to be called *before* any other function of the C-API.

Here is a minimal example of a Cython file test_gmpy2.pyx:

```
"A minimal cython file test_gmpy2.pyx"

from gmpy2 cimport *

cdef extern from "gmp.h":
    void mpz_set_si(mpz_t, long)
```

(continues on next page)

(continued from previous page)

```
import_gmpy2()   # needed to initialize the C-API

cdef mpz z = GMPy_MPZ_New(NULL)
mpz_set_si(MPZ(z), -7)

print(z + 3)
```

The corresponding setup.py is given below.

```
"A minimal setup.py for compiling test_gmpy2.pyx"

import sys

from setuptools import Extension, setup
from Cython.Build import cythonize

ext = Extension("test_gmpy2", ["test_gmpy2.pyx"],
                include_dirs=sys.path, libraries=['gmp', 'mpfr', 'mpc'])

setup(name="cython_gmpy_test",
      ext_modules=cythonize([ext], include_path=sys.path))
```

With these two files in the same repository, you should be able to compile your module using

```
$ python setup.py build_ext --inplace
```

For more about compilation and installation of cython files and extension modules, please refer to the official documentation of Cython and distutils.

CHAPTER

ELEVEN

CONVERSION METHODS AND GMPY2'S NUMBERS

11.1 Conversion methods

A python object could interact with gmpy2 if it implements one of the following methods:

- `__mpz__` : return an object of type *mpz*.
- `__mpq__` : return an object of type *mpq*.
- `__mpfr__` : return an object of type *mpfr*.
- `__mpc__` : return an object of type *mpc*.

Implementing on of these methods allow gmpy2 to convert a python object into a gmpy2 type. Example:

```
>>> from gmpy2 import mpz
>>> class CustInt:
...     def __init__(self, x):
...             self.x = x
...     def __mpz__(self):
...             return mpz(self.x)
...
>>> ci = CustInt(5)
>>> z = mpz(ci); z
mpz(5)
>>> type(z)
<class 'gmpy2.mpz'>
```

11.2 Arithmetic operations

gmpy2 allow arithmetic operations between gmpy2 numbers and objects with conversion methods. Operation with object that implements floating conversion and exact conversion methods are not supported. That means that only the following cases are supported:

- An integer type have to implement `__mpz__`
- A rational type have to implement `__mpq__` and can implement `__mpz__`
- A real type have to implement `__mpfr__`
- A complex type have to implement `__mpc__` and can implement `__mpfr__`

Examples:

```
>>> import gmpy2
>>> from gmpy2 import mpz, mpq, mpfr, mpc
>>> gmpy2.set_context(gmpy2.context())
>>> class Q:
...     def __mpz__(self): return mpz(1)
...     def __mpq__(self): return mpq(3,2)
>>> q = Q()
>>> mpz(2) + q
mpq(7,2)
>>> mpq(1,2) * q
mpq(3,4)
>>> mpfr(10) * q
mpfr('15.0')
```

CHAPTER

TWELVE

CHANGES FOR GMPY2 RELEASES

12.1 Changes in gmpy2 2.1.0rc2

- Documentation updates.
- Improvements to build environment.

12.2 Changes in gmpy2 2.1.0rc1

- **Added support for embedded underscore characters in string**
  literals.
- Allow GIL release for mpz/xmpz/mpq types only.

12.3 Changes in gmpy2 2.1.0b6

- **Improve argument type processing by saving type information to**
  decrease the number of type check calls. Especially helpful for mpfr and mpc types. (Not complete but common operations are done.)
- Resolve bug in mpfr to mpq conversion; issue #287.
- **Added limited support for releasing the GIL; disabled by default;**
  see context.allow_release_gil.
- **Refactored handling of inplace operations for mpz and xmpz types;**
  inplace operations on xmpz will only return an xmpz result.
- **Refactored handling of conversion to C integer types. Some**
  exception types changes to reflect Python types.
- **gcd() and lcm() now support more than two arguments to align with**
  the corresponding functions in the math module.

12.4 Changes in gmpy2 2.1.0b5

- **Avoid MPFR bug in mfr_fac_ui (gmpy2.factorial) on platforms where** long is 32-bits and argument is >= 44787929.
- Fixed testing bugs with Python 2.7.
- Fixed mpz(0) to C long or long long.
- Fixed incorrect results in f2q().
- Adjust test suite to reflect changes in output in MPFR 4.1.0.

12.5 Changes in gmpy2 2.1.0b4

- Fix comparisons with mpq and custom rational objects.
- Fixes for some uncommon integer conversions scenarios.

12.6 Changes in gmpy2 2.1.0b3

- Version bump only.

12.7 Changes in gmpy2 2.1.0b2

- Many bug fixes.

12.8 Changes in gmpy2 2.1.0b1

- Added cmp() and cmp_abs().
- Improved compatibility with _numbers_ protocol.
- Many bug fixes.

12.9 Changes in gmpy2 2.1.a05

- Fix qdiv() not returning mpz() when it should.
- Added root_of_unity().

12.10 Changes in gmpy2 2.1.0a4

- Fix issue 204; missing file for Cython.
- **Additional support for MPFR 4**
  - Add fmma() and fmms()

12.11 Changes in gmpy2 2.1.0a3

- Updates to setup.py.
- **Initial support for MPFR4**
  - Add nrandom()
  - grandom() now calls nrandom twice; may return different values versus MPFR3
  - Add rootn(); same as root() except different sign when taking even root of -0.0

12.12 Changes in gmpy2 2.1.0a2

- Revised build process.
- Removal of unused code/macros.
- Cleanup of Cython interface.

12.13 Changes in gmpy2 2.1.0a1

- Thread-safe contexts are now supported. Properly integrating thread-safe contexts required an extensive rewrite of almost all internal functions.
- MPFR and MPC are now required. It is no longer possible to build a version of gmpy2 that only supports the GMP library.
- The function inverse() now raises an exception if the inverse does not exist.
- Context methods have been added for MPFR/MPC related functions.
- A new context option (*rational_division*) has been added that changes the behavior of integer division involving *mpz* instances to return a rational result instead of a floating point result.
- gmpy2 types are now registered in the numeric tower.
- In previous versions of gmpy2, *gmpy2.mpz* was a factory function that returned an *mpz* instance. *gmpy2.mpz* is now an actual type. The same is true for the other gmpy2 types.
- If a Python object has an __mpz__ method, it will be called bye *mpz()* to allow an unrecognized type to be converted to an mpz instance. The same is true for the other gmpy2 types.
- A new C-API and Cython interface has been added.

12.14 Changes in gmpy2 2.0.4

- Fix bit_scan0() for negative values.
- Changes to setup.py to allow static linking.
- Fix performance regression with mpmath and Python 3.

12.15 Changes in gmpy2 2.0.3

- Fix lucas2() and atanh(); they were returning incorrect values.

12.16 Changes in gmpy2 2.0.2

- Rebuild Windows binary installers due to MPIR 2.6.0 bug in next_prime().
- Another fix for is_extra_strong_lucas_prp().

12.17 Changes in gmpy2 2.0.1

- Updated setup.py to work in more situations.
- Corrected exception handling in basic operations with mpfr type.
- Correct InvalidOperation exception not raised in certain circumstances.
- invert() now raises an exception if the modular inverse does not exist.
- Fixed internal exception in is_bpsw_prp() and is_strong_bpsw_prp().
- Updated is_extra_strong_lucas_prp() to latest version.

12.18 Changes in gmpy2 2.0.0

- Fix segmentation fault in _mpmath_normalize (an undocumented helper function specifically for mpmath.)
- Improved setup.py See below for documentation on the changes.
- Fix issues when compiled without support for MPFR.
- Conversion of too large an mpz to float now raises OverflowError instead of returning *inf*.
- Renamed min2()/max2() to minnum()/maxnum()
- The build and install process (i.e. setup.py) has been completely rewritten. See the Installation section for more information.
- get_context() no longer accepts keyword arguments.

12.19 Known issues in gmpy2 2.0.0

- The test suite is still incomplete.

12.20 Changes in gmpy2 2.0.0b4

- Added __ceil__, __floor__, __trunc__, and __round__ methods to mpz and mpq types.
- Added __complex__ to mpc type.
- round(mpfr) now correctly returns an mpz type.
- If no arguments are given to mpz, mpq, mpfr, mpc, and xmpz, return 0 of the appropriate type.
- Fix broken comparison between mpz and mpq when mpz is on the left.
- Added __sizeof__ to all types. *Note: sys.getsizeof() calls __sizeof__ to get the memory size of a gmpy2 object. The returned value reflects the size of the allocated memory which may be larger than the actual minimum memory required by the object.*

12.21 Known issues in gmpy2 2.0.0b4

- The new test suite (test/runtest.py) is incomplete and some tests fail on Python 2.x due to formatting issues.

12.22 Changes in gmpy2 2.0.0b3

- mp_version(), mpc_version(), and mpfr_version() now return normal strings on Python 2.x instead of Unicode strings.
- Faster conversion of the standard library Fraction type to mpq.
- Improved conversion of the Decimal type to mpfr.
- Consistently return OverflowError when converting "inf".
- Fix mpz.__format__() when the format code includes "#".
- Add is_infinite() and deprecate is_inf().
- Add is_finite() and deprecate is_number().
- Fixed the various is_XXX() tests when used with mpc.
- Added caching for mpc objects.
- Faster code path for basic operation is both operands are mpfr or mpc.
- Fix mpfr + float segmentation fault.

12.23 Changes in gmpy2 2.0.0b2

- Allow xmpz slice assignment to increase length of xmpz instance by specifying a value for stop.
- Fixed reference counting bug in several is_xxx_prp() tests.
- Added iter_bits(), iter_clear(), iter_set() methods to xmpz.
- Added powmod() for easy access to three argument pow().
- Removed addmul() and submul() which were added in 2.0.0b1 since they are slower than just using Python code.
- Bug fix in gcd_ext when both arguments are not mpz.
- Added ieee() to create contexts for 32, 64, or 128 bit floats.
- Bug fix in context() not setting emax/emin correctly if they had been changed earlier.
- Contexts can be directly used in with statement without requiring set_context()/local_context() sequence.
- local_context() now accepts an optional context.

12.24 Changes in gmpy2 2.0.0b1 and earlier

- Renamed functions that manipulate individual bits to bit_XXX() to align with bit_length().
- Added caching for mpq.
- Added rootrem(), fib2(), lucas(), lucas2().
- Support changed hash function in Python 3.2.
- Added is_even(), is_odd().
- Add caching of the calculated hash value.
- Add xmpz (mutable mpz) type.
- Fix mpq formatting issue.
- Add read/write bit access using slices to xmpz.
- Add read-only bit access using slices to mpz.
- Add pack()/unpack() methods to split/join an integer into n-bit chunks.
- Add support for MPFR (casevh)
- Removed fcoform float conversion modifier.
- Add support for MPC.
- Added context manager.
- Allow building with just GMP/MPIR if MPFR not available.
- Allow building with GMP/MPIR and MPFR if MPC not available.
- Removed most instance methods in favor of gmpy2.function. The general guideline is that *properties* of an instance can be done via instance methods but *functions* that return a new result are done using gmpy2.function.
- Added __ceil__, __floor__, and __trunc__ methods since they are called by math.ceil(), math.floor(), and math.trunc().
- Removed gmpy2.pow() to avoid conflicts.

- Removed gmpy2._copy and added xmpz.copy.
- Added support for __format__.
- Added as_integer_ratio, as_mantissa_exp, as_simple_fraction.
- Updated rich_compare.
- Require MPFR 3.1.0+ to get divby0 support.
- Added fsum(), degrees(), radians().
- Updated random number generation support.
- Changed license to LGPL 3+.
- Added lucasu, lucasu_mod, lucasv, and lucasv_mod. *Based on code contributed by David Cleaver.*
- Added probable-prime tests. *Based on code contributed by David Cleaver.*
- Added to_binary()/from_binary.
- Renamed numdigits() to num_digits().
- Added keyword precision to constants.
- Added addmul() and submul().
- Added __round__(), round2(), round_away() for mpfr.
- round() is no longer a module level function.
- Renamed module functions min()/max() to min2()/max2().
- No longer conflicts with builtin min() and max()
- Removed set_debug() and related functionality.

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X

Y

Z