gmpy2 Documentation

Release 2.0.0b3

Case Van Horsen

December 13, 2012

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

- MPIR is based on the GMP library but adds support for Microsoft's Visual Studio compiler. It is used to create the Windows binaries.

  Home page: http://www.mpir.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 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.

1.2 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.

1.3 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.

1.4 Installing gmpy2 on Windows

Pre-compiled versions of gmpy2 are available at Downloads . Please select the installer that corresponds to the version of Python installed on your computer. Note that either a 32 or 64-bit version of Python can be installed on a 64-bit version of Windows. If you get an error message stating that Python could not be found in the registry, you have the wrong version of the gmpy2 installer.

1.5 Installing gmpy2 on Unix/Linux

1.5.1 Requirements

gmpy2 has only been tested with the most recent versions of GMP, MPFR and MPC. Specifically, for integer and rational support, gmpy2 requires GMP 5.0.x or later. To support multiple-precision floating point arithmetic, MPFR

3.1.x or later is required. MPC 1.0.1 or later is required for complex arithmetic.

The MPC and MPFR libraries are optional. If the MPC library is not available, gmpy2 will still support integer, rational, and real floating-point arithmetic. If the MPFR library is not available, gmpy2 will only support integer and rational arithmetic. The mpf type included with GMP is no longer supported.

1.5.2 Short Instructions

If your system includes sufficiently recent versions of GMP, MPFR and MPC, and you have the development libraries installed, compiling should be as simple as:

```
cd <gmpy2 source directory>
python setup.py install
```

If this fails, read on.

1.5.3 Detailed Instructions

If your Linux distribution does not support recent versions of GMP, MPFR and MPC, you will need to compile your own versions. To avoid any possible conflict with existing libraries on your system, the following instructions install GMP, MPFR and MPC in a separate directory. The examples use /opt/local but you can use another directory if you choose.

Create the desired destination directory for GMP, MPFR, and MPC.

```
$ mkdir /opt/local
```

Download and un-tar the GMP source code. Change to GMP source directory and compile GMP.

```
$ cd /opt/local/src/gmp-5.0.2
$ ./configure --prefix=/opt/local
$ make
$ make check
$ make install
```

Download and un-tar the MPFR source code. Change to MPFR source directory and compile MPFR.

```
$ cd /opt/local/mpfr-3.1.1
$ ./configure --prefix=/opt/local --with-gmp=/opt/local
$ make
$ make check
$ make install
```

Download and un-tar the MPC source code. Change to MPC source directory and compile MPC.

```
$ cd /opt/local/mpc-1.0.1
$ ./configure --prefix=/opt/local --with-gmp=/opt/local --with-mpfr=/opt/local
$ make
$ make check
$ make install
```

Compile gmpy2 and specify the location of GMP, MPFR and MPC.

```
$ python setup.py build_ext -Ddir=/opt/local install
```

If you get a “permission denied” error message, you may need to use:

```
$ sudo python setup.py build_ext -Ddir=/home/opt/local install
```

1.6 Miscellaneous gmpy2 Functions

from_binary(...) from_binary(bytes) returns a gmpy2 object from a byte sequence created by to_binary().

get_cache(...) get_cache() returns the current cache size (number of objects) and the maximum size per object (number of limbs).

gmpy2 maintains an internal list of freed *mpz*, *xmpz*, *mpq*, *mpfr*, and *mpc* objects for reuse. The cache significantly improves performance but also increases the memory footprint.

license(...) license() returns the gmpy2 license information.

mp_limbsize(...) mp_limbsize() returns the number of bits per limb used by the GMP or MPIR libarary.

mp_version(...) mp_version() returns the version of the GMP or MPIR library.

mpc_version(...) mpc_version() returns the version of the MPC library.

mpfr_version(...) mpfr_version() returns the version of the MPFR library.

random_state(...) random_state([seed]) returns a new object containing state information for the random number generator. An optional integer argument can be specified as the seed value. Only the Mersenne Twister random number generator is supported.

set_cache(...) set_cache(number, size) updates the maximum number of freed objects of each type that are cached and the maximum size (in limbs) of each object. The maximum number of objects of each type that can be cached is 1000. The maximum size of an object is 16384. The maximum size of an object is approximately 64K on 32-bit systems and 128K on 64-bit systems.

Note: The caching options are global to gmpy2. Changes are not thread-safe. A change in one thread will impact all threads.

---

to_binary(...) to_binary(x) returns a byte sequence from a gmpy2 object. All object types are supported.

version(...) version() returns the version of gmpy2.

CHAPTER

TWO

MULTIPLE-PRECISION INTEGERS

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

2.1 Examples

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

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.

2.2 mpz Methods

bit_clear(...) x.bit_clear(n) returns a copy of *x* with bit *n* set to 0.

bit_flip(...) x.bit_flip(n) returns a copy of *x* with bit *n* inverted.

bit_length(...) x.bit_length() returns the number of significant bits in the radix-2 representation of *x*. For compatibility with Python, mpz(0).bit_length() returns 0.

bit_scan0(...) x.bit_scan0(n) returns the index of the first 0-bit of *x* with index >= *n*. 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. *n* must be >= 0.

bit_scan1(...) x.bit_scan1(n) returns the index of the first 1-bit of *x* with index >= *n*. 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. *n* must be >= 0.

bit_set(...) x.bit_set(n) returns a copy of *x* with bit *n* set to 0.

bit_test(...) x.bit_test(n) returns True if bit *n* of *x* is set, and False if it is not set.

digits(...) x.digits([base=10]) returns a string representing *x* in radix *base*.

num_digits(...) x.num_digits([base=10]) returns the length of the string representing the absolute value of *x* in radix *base*. The result is correct if base is a power of 2. For other other bases, the result is usually correct but may be 1 too large. *base* can range between 2 and 62, inclusive.

2.3 mpz Functions

add(...) add(x, y) returns *x* + *y*. The result type depends on the input types.

bincoef(...) bincoef(x, n) returns the binomial coefficient. *n* must be >= 0.

bit_clear(...) bit_clear(x, n) returns a copy of *x* with bit *n* set to 0.

bit_flip(...) bit_flip(x, n) returns a copy of *x* with bit *n* inverted.

bit_length(...) bit_length(x) returns the number of significant bits in the radix-2 representation of *x*. For compatibility with Python, mpz(0).bit_length() returns 0 while mpz(0).num_digits(2) returns 1.

bit_mask(...) bit_mask(n) returns an *mpz* object exactly *n* bits in length with all bits set.

bit_scan0(...) bit_scan0(x, n) returns the index of the first 0-bit of *x* with index >= *n*. 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. *n* must be >= 0.

bit_scan1(...) bit_scan1(x, n) returns the index of the first 1-bit of *x* with index >= *n*. 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. *n* must be >= 0.

bit_set(...) bit_set(x, n) returns a copy of *x* with bit *n* set to 0.

bit_test(...) bit_test(x, n) returns True if bit *n* of *x* is set, and False if it is not set.

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

c_div_2exp(...) c_div_2exp(x, n) returns the quotient of *x* divided by 2**n. The quotient is rounded towards +Inf (ceiling rounding). *x* must be an integer and *n* must be > 0.

c_divmod(...) c_divmod(x, y) returns 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.

c_divmod_2exp(...) c_divmod_2exp(x ,n) returns the quotient and remainder of *x* divided by 2**n. The quotient is rounded towards +Inf (ceiling rounding) and the remainder will be negative or zero. *x* must be an integer and *n* must be > 0.

c_mod(...) c_mod(x, y) returns the remainder of *x* divided by *y*. The remainder will have the opposite sign of *y*. *x* and *y* must be integers.

c_mod_2exp(...) c_mod_2exp(x, n) returns the remainder of *x* divided by 2**n. The remainder will be negative. *x* must be an integer and *n* must be > 0.

comb(...) comb(x, n) returns the number of combinations of *x* things, taking *n* at a time. *n* must be >= 0.

digits(...) digits(x[, base=10]) returns a string representing *x* in radix *base*.

div(...) div(x, y) returns *x* / *y*. The result type depends on the input types.

divexact(...) divexact(x, y) returns the quotient of *x* divided by *y*. Faster than standard division but requires the remainder is zero!

divm(...) divm(a, b, m) returns *x* such that *b* * *x* == *a* modulo *m*. Raises a ZeroDivisionError exception if no such value *x* exists.

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

f_div_2exp(...) f_div_2exp(x, n) returns the quotient of *x* divided by 2**n. The quotient is rounded towards -Inf (floor rounding). *x* must be an integer and *n* must be > 0.

f_divmod(...) f_divmod(x, y) returns 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.

f_divmod_2exp(...) f_divmod_2exp(x, n) returns 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 and *n* must be > 0.

f_mod(...) f_mod(x, y) returns the remainder of *x* divided by *y*. The remainder will have the same sign as *y*. *x* and *y* must be integers.

f_mod_2exp(...) f_mod_2exp(x, n) returns remainder of *x* divided by 2**n. The remainder will be positive. *x* must be an integer and *n* must be > 0.

fac(...) fac(n) returns the exact factorial of *n*. Use factorial() to get the floating-point approximation.

fib(...) fib(n) returns the *n*-th Fibonacci number.

fib2(...) fib2(n) returns a 2-tuple with the (*n*-1)-th and *n*-th Fibonacci numbers.

gcd(...) gcd(a, b) returns the greatest common denominator of integers *a* and *b*.

gcdext(...) gcdext(a, b) returns a 3-element tuple (*g*, *s*, *t*) such that

g == gcd(*a*, *b*) and *g* == *a* * *s* + *b* * *t*

hamdist(...) hamdist(x, y) returns the Hamming distance (number of bit-positions where the bits differ) between integers *x* and *y*.

invert(...) invert(x, m) returns *y* such that *x* * *y* == 1 modulo *m*, or 0 if no such *y* exists.

iroot(...) iroot(x,n) returns a 2-element tuple (*y*, *b*) such that *y* is the integer *n*-th root of *x* and *b* is True if the root is exact. *x* must be >= 0 and *n* must be > 0.

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

is_even(...) is_even(x) returns True if *x* is even, False otherwise.

is_odd(...) is_odd(x) returns True if *x* is odd, False otherwise.

is_power(...) is_power(x) returns True if *x* is a perfect power, False otherwise.

is_prime(...) is_prime(x[, n=25]) returns True if *x* is **probably** prime. False is returned if *x* is definately composite. *x* is checked for small divisors and up to *n* Miller-Rabin tests are performed. The actual tests performed may vary based on version of GMP or MPIR used.

is_square(...) is_square(x) returns True if *x* is a perfect square, False otherwise.

isqrt(...) isqrt(x) returns the integer square root of an integer *x*. *x* must be >= 0.

isqrt_rem(...) isqrt_rem(x) returns a 2-tuple (*s*, *t*) such that *s* = isqrt(*x*) and *t* = *x* - *s* * *s*. *x* must be >= 0.

jacobi(...) jacobi(x, y) returns the Jacobi symbol (*x* | *y*). *y* must be odd and > 0.

kronecker(...) kronecker(x, y) returns the Kronecker-Jacobi symbol (*x* | *y*).

lcm(...) lcm(a, b) returns the lowest common multiple of integers *a* and *b*.

legendre(...) legendre(x, y) returns the Legendre symbol (*x* | *y*). *y* is assumed to be an odd prime.

lucas(...) lucas(n) returns the *n*-th Lucas number.

lucas2(...) lucas2(n) returns a 2-tuple with the (*n*-1)-th and *n*-th Lucas numbers.

mpz(...) mpz(n) returns a new *mpz* object from a numeric value *n*. If *n* is not an integer, it will be truncated to an integer.

mpz(s[, base=0]) returns a new *mpz* object from a string *s* made of digits in the given base. If base = 0, thn binary, octal, or hex Python strings are recognized by leading 0b, 0o, or 0x characters. Otherwise the string is assumed to be decimal. Values for base can range between 2 and 62.

mpz_random(...) mpz_random(random_state, n) returns a uniformly distributed random integer between 0 and *n*-1. The parameter *random_state* must be created by random_state() first.

mpz_rrandomb(...) mpz_rrandomb(random_state, b) returns a random integer between 0 and 2**b - 1 with long sequences of zeros and one in its binary representation. The parameter *random_state* must be created by random_state() first.

mpz_urandomb(...) mpz_urandomb(random_state, b) returns a uniformly distributed random integer between 0 and 2**b - 1. The parameter *random_state* must be created by random_state() first.

mul(...) mul(x, y) returns *x* * *y*. The result type depends on the input types.

next_prime(...) next_prime(x) returns the next **probable** prime number > *x*.

num_digits(...) num_digits(x[, base=10]) returns the length of the string representing the absolute value of *x* in radix *base*. The result is correct if base is a power of 2. For other other bases, the result is usually correct but may be 1 too large. *base* can range between 2 and 62, inclusive.

popcount(...) popcount(x) returns the number of bits with value 1 in *x*. If *x* < 0, the number of bits with value 1 is infinite so -1 is returned in that case.

powmod(...) powmod(x, y, m) returns (*x* ** *y*) mod *m*. The exponenent *y* can be negative, and the correct result will be returned if the inverse of *x* mod *m* exists. Otherwise, a ValueError is raised.

remove(...) remove(x, f) will remove the factor *f* from *x* as many times as possible and return a 2-tuple (*y*, *m*) where *y* = *x* // (*f* ** *m*). *f* does not divide *y*. *m* is the multiplicity of the factor *f* in *x*. *f* must be > 1.

sub(...) sub(x, y) returns *x* - *y*. The result type depends on the input types.

t_div(...) t_div(x, y) returns the quotient of *x* divided by *y*. The quotient is rounded towards zero (truncation). *x* and *y* must be integers.

t_div_2exp(...) t_div_2exp(x, n) returns the quotient of *x* divided by 2**n. The quotient is rounded towards zero (truncation). *n* must be > 0.

t_divmod(...) t_divmod(x, y) returns 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.

t_divmod_2exp(...) t_divmod_2exp(x, n) returns 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 and *n* must be > 0.

t_mod(...) t_mod(x, y) returns the remainder of *x* divided by *y*. The remainder will have the same sign as *x*. *x* and *y* must be integers.

t_mod_2exp(...) t_mod_2exp(x, n) returns the remainder of *x* divided by 2**n. The remainder will have the same sign as *x*. *x* must be an integer and *n* must be > 0.

CHAPTER

THREE

MULTIPLE-PRECISION INTEGERS (ADVANCED TOPICS)

3.1 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 orignal object and do not create a new object. Instances of *xmpz* cannot be used as dictionary keys.

```
>>> import gmpy2
>>> 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:

```
>>> 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.

```
from __future__ import print_function
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()
    result = list(sieve())
    print(time.time() - start)
    print(len(result))
```

3.2 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

is_bpsw_prp(...) is_bpsw_prp(n) will 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.

is_euler_prp(...) is_euler_prp(n,a) will 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)

is_extra_strong_lucas_prp(...) is_extra_strong_lucas_prp(n,p) will 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

is_fermat_prp(...) is_fermat_prp(n,a) will 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)

is_fibonacci_prp(...) is_fibonacci_prp(n,p,q) will return True if *n* is an 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).

is_lucas_prp(...) is_lucas_prp(n,p,q) will 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)

is_selfridge_prp(...) is_selfridge_prp(n) will 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. Let p=1 and q = (1-D)/4 and then perform a Lucas probable prime test.

is_strong_bpsw_prp(...) is_strong_bpsw_prp(n) will 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 2 and the is_strongselfridge_prp() test.

is_strong_lucas_prp(...) is_strong_lucas_prp(n,p,q) will 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

is_strong_prp(...) is_strong_prp(n,a) will return True if *n* is an 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.

is_strong_selfridge_prp(...) is_strong_selfridge_prp(n) will 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. Let p=1 and q = (1-D)/4 and then perform a strong Lucas probable prime test.

lucasu(...) lucasu(p,q,k) will 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.

lucasu_mod(...) lucasu_mod(p,q,k,n) will 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.

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

lucasv_mod(...) lucasv_mod(p,q,k,n) will return the k-th element of the Lucas V sequence defined by parameters (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

FOUR

MULTIPLE-PRECISION RATIONALS

gmpy2 provides a rational type call *mpq*. It should be a replacement for Python's fractions.Fraction module.

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

4.1 mpq Methods

digits(...) x.digits([base=10]) returns 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.

4.2 mpq Attributes

denominator x.denomintor returns the denominator of *x*.

numerator x.numerator returns the numerator of *x*.

4.3 mpq Functions

add(...) add(x, y) returns *x* + *y*. The result type depends on the input types.

div(...) div(x, y) returns *x* / *y*. The result type depends on the input types.

f2q(...) f2q(x[, err]) returns the best *mpq* approximating *x* to within relative error *err*. Default is the precision of *x*. If *x* is not an *mpfr*, it is converted to an *mpfr*. Uses Stern-Brocot tree to find the best approximation. An *mpz* is returned if the the denominator is 1. If *err* < 0, then the relative error sought is 2.0 ** *err*.

mpq(...) mpq(n) returns an *mpq* object with a numeric value *n*. Decimal and Fraction values are converted exactly.

mpq(n, m) returns an *mpq* object with a numeric value *n* / *m*.

mpq(s[, base=10]) returns an *mpq* object from a string *s* made up of digits in the given base. *s* may be made up of two numbers in the same base separated by a '/' character. If *base* == 10, then an embedded '.' indicates a number with a decimal fractional part.

mul(...) mul(x, y) returns *x* * *y*. The result type depends on the input types.

qdiv(...) qdiv(x[, y=1]) returns *x/y* as *mpz* if possible, or as *mpq* if *x* is not exactly divisible by *y*.

sub(...) sub(x, y) returns *x* - *y*. The result type depends on the input types.

CHAPTER

FIVE

MULTIPLE-PRECISION REALS

gmpy2 replaces the *mpf* type from gmpy 1.x with a new *mpfr* type 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 precison 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
>>> 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)
>>>
```

5.1 Contexts

Warning: Contexts and context managers are not thread-safe! Modifying the context in one thread will impact all other threads.

A *context* 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.

`gmpy2.context()` creates a new context with all options set to default. `gmpy2.set_context(ctx)` will set the active context to *ctx*. `gmpy2.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 following example just modifies the precision. The remaining options will be discussed later.

```
>>> 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,
        trap_expbound=False,
        allow_complex=False)
>>> gmpy2.sqrt(5)
mpfr('2.2360679774997898')
>>> gmpy2.get_context().precision=100
>>> gmpy2.sqrt(5)
mpfr('2.23606797749978969640917366873127',100)
>>> gmpy2.get_context().precision+=20
>>> gmpy2.sqrt(5)
mpfr('2.23606797749978969640917366873127623510',120)
>>> ctx=gmpy2.get_context()
>>> ctx.precision+=20
>>> gmpy2.sqrt(5)
mpfr('2.23606797749978969640917366873127623544061823',140)
>>> gmpy2.set_context(gmpy2.context())
>>> gmpy2.sqrt(5)
mpfr('2.2360679774997898')
>>> ctx.precision+=20
>>> gmpy2.sqrt(5)
mpfr('2.2360679774997898')
>>> gmpy2.set_context(ctx)
>>> gmpy2.sqrt(5)
mpfr('2.23606797749978969640917366873127623544061835961153',160)
>>>
```

5.2 Context Attributes

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.

---

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.

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.

round There are five rounding modes availble to *mpfr* types:

`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.

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*.

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*.

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()**.

Note: It is possible to change the values of emin/emax such that previous *mpfr* values are no longer valid numbers but should either underflow to +/-0.0 or overflow to +/-Infinity. To raise an exception if this occurs, see **trap_expbound**.

---

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_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`.

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.

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

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.

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

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.

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

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.

```
>>> gmpy2.set_context(gmpy2.context())
>>> mpfr('nan') == mpfr('nan')
False
>>> gmpy2.get_context().trap_erange=True
>>> mpfr('nan') == mpfr('nan')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
gmpy2.RangeError: comparison with NaN
>>>
```

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

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`.

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

trap_expbound This attribute controls whether or not an `ExponentOutOfBoundsError` exception is raised if exponents in an operand are outside the current emin/emax limits.

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

5.3 Context Methods

clear_flags() Clear the underflow, overflow, inexact, invalid, erange, and divzero flags.

5.4 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.

`gmpy2.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 orginal active context is restored when the block completes.

In the following example, the current context is saved by `gmpy2.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.

```
>>> 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
>>>
```

A context object can also be used directly to create a context manager block. However, instead of restoring the context to the active context when the `with ...` statement is executed, the restored context is the context used before any keyword argument modifications.

The code:

:: with gmpy2.ieee(64) as ctx:

is equivalent to:

:: gmpy2.set_context(gmpy2.ieee(64)) with gmpy2.local_context() as ctx:

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

5.5 mpfr Methods

as_integer_ratio() Returns a 2-tuple containing the numerator and denominator after converting the *mpfr* object into the exact rational equivalent. The return 2-tuple is equivalent to Python's as_integer_ratio() method of built-in float objects.

as_mantissa_exp() Returns a 2-tuple containing the mantissa and exponent.

as_simple_fraction() Returns an *mpq* containing the simpliest rational value that approximates the *mpfr* value with an error less than 1/(2**precision).

conjugate() Returns the complex conjugate. For *mpfr* objects, returns a copy of the original object.

digits() Returns a 3-tuple containing the mantissa, the exponent, and the number of bits of precision. The mantissa is represented as a string in the specified base with up to 'prec' digits. 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.

is_integer() Returns True if the *mpfr* object is an integer.

5.6 mpfr Attributes

imag Returns the imaginary component. For *mpfr* objects, returns 0.

precision Returns the precision of the *mpfr* object.

rc The result code (also known as ternary value in the MPFR documentation) is 0 if the value of the *mpfr* object is exactly equal to the exact, infinite precision value. If the result code is 1, then the value of the *mpfr* object is greater than the exact value. If the result code is -1, then the value of the *mpfr* object is less than the exact, infinite precision value.

real Returns the real component. For *mpfr* objects, returns a copy of the original object.

5.7 mpfr Functions

acos(...) acos(x) returns the arc-cosine of x. x is measured in radians. If context.allow_complex is True, then an *mpc* result will be returned for abs(x) > 1.

acosh(...) acosh(x) returns the inverse hyperbolic cosine of x.

add(...) add(x, y) returns x + y. The type of the result is based on the types of the arguments.

agm(...) agm(x, y) returns the arithmetic-geometric mean of x and y.

ai(...) ai(x) returns the Airy function of x.

asin(...) asin(x) returns the arc-sine of x. x is measured in radians. If context.allow_complex is True, then an *mpc* result will be returned for abs(x) > 1.

asinh(...) asinh(x) return the inverse hyperbolic sine of x.

atan(...) atan(x) returns the arc-tangent of x. x is measured in radians.

atan2(...) atan2(y, x) returns the arc-tangent of (y/x).

atanh(...) atanh(x) returns the inverse hyperbolic tangent of x. If context.allow_complex is True, then an *mpc* result will be returned for abs(x) > 1.

cbrt(...) cbrt(x) returns the cube root of x.

ceil(...) ceil(x) returns the 'mpfr' that is the smallest integer >= x.

check_range(...) check_range(x) return a new 'mpfr' with exponent that lies within the current range of emin and emax.

const_catalan(...) const_catalan([precision=0]) returns the catalan constant using the specified precision. If no precision is specified, the default precision is used.

const_euler(...) const_euler([precision=0]) returns the euler constant using the specified precision. If no precision is specified, the default precision is used.

const_log2(...) const_log2([precision=0]) returns the log2 constant using the specified precision. If no precision is specified, the default precision is used.

const_pi(...) const_pi([precision=0]) returns the constant pi using the specified precision. If no precision is specified, the default precision is used.

context(...) context() returns a new context manager controlling MPFR and MPC arithmetic.

cos(...) cos(x) seturns the cosine of x. x is measured in radians.

cosh(...) cosh(x) returns the hyperbolic cosine of x.

cot(...) cot(x) returns the cotangent of x. x is measured in radians.

coth(...) coth(x) returns the hyperbolic cotangent of x.

csc(...) csc(x) returns the cosecant of x. x is measured in radians.

csch(...) csch(x) returns the hyperbolic cosecant of x.

degrees(...) degrees(x) converts an angle measurement x from radians to degrees.

digamma(...) digamma(x) returns the digamma of x.

div(...) div(x, y) returns x / y. The type of the result is based on the types of the arguments.

div_2exp(...) div_2exp(x, n) returns an 'mpfr' or 'mpc' divided by 2**n.

eint(...) eint(x) returns the exponential integral of x.

erf(...) erf(x) returns the error function of x.

erfc(...) erfc(x) returns the complementary error function of x.

exp(...) exp(x) returns e**x.

exp10(...) exp10(x) returns 10**x.

exp2(...) exp2(x) returns 2**x.

expm1(...) expm1(x) returns e**x - 1. expm1() is more accurate than exp(x) - 1 when x is small.

f2q(...) f2q(x[,err]) returns the simplest *mpq* approximating x to within relative error err. Default is the precision of x. Uses Stern-Brocot tree to find the simplist approximation. An *mpz* is returned if the the denominator is 1. If err<0, error sought is 2.0 ** err.

factorial(...) factorial(n) returns the floating-point approximation to the factorial of n.

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

floor(...) floor(x) returns the 'mpfr' that is the smallest integer <= x.

fma(...) fma(x, y, z) returns correctly rounded result of (x * y) + z.

fmod(...) fmod(x, y) returns x - n*y where n is the integer quotient of x/y, rounded to 0.

fms(...) fms(x, y, z) returns correctly rounded result of (x * y) - z.

frac(...) frac(x) returns the fractional part of x.

frexp(...) frexp(x) returns a tuple containing the exponent and mantissa of x.

fsum(...) fsum(iterable) returns the accurate sum of the values in the iterable.

gamma(...) gamma(x) returns the gamma of x.

get_exp(...) get_exp(mpfr) returns the exponent of an *mpfr*. Returns 0 for NaN or Infinity and sets the erange flag and will raise an exception if trap_erange is set.

hypot(...) hypot(y, x) returns square root of (x**2 + y**2).

ieee(...) ieee(bitwidth) returns a context with settings for 32-bit (aka single), 64-bit (aka double), or 128-bit (aka quadruple) precision floating point types.

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

is_finite(...) is_finite(x) returns True if x is an actual number (i.e. not NaN or Infinity).

is_inf(...) is_inf(x) returns True if x is Infinity or -Infinity.

Note: **is_inf()** is deprecated; please use **if_infinite()**.

---

is_infinite(...) is_infinite(x) returns True if x Infinity or -Infinity.

is_nan(...) is_nan(x) returns True if x is NaN (Not-A-Number).

is_number(...) is_number(x) returns True if x is an actual number (i.e. not NaN or Infinity).

Note: **is_number()** is deprecated; please use **is_finite()**.

---

is_regular(...) is_regular(x) returns True if x is not zero, NaN, or Infinity.

is_signed(...) is_signed(x) returns True if the sign bit of x is set.

is_unordered(...) is_unordered(x,y) returns True if either x and/or y is NaN.

is_zero(...) is_zero(x) returns True if x is zero.

j0(...) j0(x) returns the Bessel function of the first kind of order 0 of x.

j1(...) j1(x) returns the Bessel function of the first kind of order 1 of x.

jn(...) jn(x,n) returns the Bessel function of the first kind of order n of x.

lgamma(...) lgamma(x) returns a tuple containing the logarithm of the absolute value of gamma(x) and the sign of gamma(x)

li2(...) li2(x) returns the real part of dilogarithm of x.

lngamma(...) lngamma(x) returns the logarithm of gamma(x).

log(...) log(x) returns the natural logarithm of x.

log10(...) log10(x) returns the base-10 logarithm of x.

log1p(...) log1p(x) returns the natural logarithm of (1+x).

log2(...) log2(x) returns the base-2 logarithm of x.

max2(...) max2(x, y) returns the maximum of x and y. The result may be rounded to match the current context. Use the builtin max() to get an exact copy of the largest object without any rounding.

min2(...) min2(x, y) returns the minimum of x and y. The result may be rounded to match the current context. Use the builtin min() to get an exact copy of the smallest object without any rounding.

modf(...) modf(x) returns a tuple containing the integer and fractional portions of x.

mpfr(...) mpfr(n[, precison=0]) returns an *mpfr* object after converting a numeric value n. If no precision, or a precision of 0, is specified; the precision is taken from the current context.

mpfr(s[, precision=0[, [base=0]]) returns an *mpfr* object after converting a string 's' made up of digits in the given base, possibly with fractional part (with period as a separator) and/or exponent (with exponent marker 'e' for base<=10, else '@'). If no precision, or a precision of 0, is specified; the precison is taken from the current context. The base of the string representation must be 0 or in the interval 2 ... 62. If the base is 0, the leading digits of the string are used to identify the base: 0b implies base=2, 0x implies base=16, otherwise base=10 is assumed.

mpfr_from_old_binary(...) mpfr_from_old_binary(string) returns an *mpfr* from a GMPY 1.x binary mpf format. Please use to_binary()/from_binary() to convert GMPY2 objects to or from a binary format.

mpfr_grandom(...) mpfr_grandom(random_state) returns two random numbers with gaussian distribution. The parameter *random_state* must be created by random_state() first.

mpfr_random(...) mpfr_random(random_state) returns a uniformly distributed number between [0,1]. The parameter *random_state* must be created by random_state() first.

mul(...) mul(x, y) returns x * y. The type of the result is based on the types of the arguments.

mul_2exp(...) mul_2exp(x, n) returns 'mpfr' or 'mpc' multiplied by 2**n.

nan(...) nan() returns an 'mpfr' initialized to NaN (Not-A-Number).

next_above(...) next_above(x) returns the next 'mpfr' from x toward +Infinity.

next_below(...) next_below(x) returns the next 'mpfr' from x toward -Infinity.

radians(...) radians(x) converts an angle measurement x from degrees to radians.

rec_sqrt(...) rec_sqrt(x) returns the reciprocal of the square root of x.

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

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

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

rint(...) rint(x) returns x rounded to the nearest integer using the current rounding mode.

rint_ceil(...) rint_ceil(x) returns 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.

rint_floor(...) rint_floor(x) returns 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.

rint_round(...) rint_round(x) returns 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.

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

root(...) root(x, n) returns n-th root of x. The result always an *mpfr*.

round2(...) round2(x[, n]) returns x rounded to n bits. Uses default precision if n is not specified. See round_away() to access the mpfr_round() function. Use the builtin round() to round x to n decimal digits.

round_away(...) round_away(x) returns an *mpfr* by rounding x the nearest integer, with ties rounded away from 0.

sec(...) sec(x) returns the secant of x. x is measured in radians.

sech(...) sech(x) returns the hyperbolic secant of x.

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

set_sign(...) set_sign(x, bool) returns a copy of x with it's sign bit set if *bool* evaluates to True.

sign(...) sign(x) returns -1 if x < 0, 0 if x == 0, or +1 if x >0.

sin(...) sin(x) returns the sine of x. x is measured in radians.

sin_cos(...) sin_cos(x) returns a tuple containing the sine and cosine of x. x is measured in radians.

sinh(...) sinh(x) returns the hyberbolic sine of x.

sinh_cosh(...) sinh_cosh(x) returns a tuple containing the hyperbolic sine and cosine of x.

sqrt(...) sqrt(x) returns the square root of x. If x is integer, rational, or real, then an *mpfr* will be returned. If x is complex, then an *mpc* will be returned. If context.allow_complex is True, negative values of x will return an *mpc*.

square(...) square(x) returns x * x. The type of the result is based on the types of the arguments.

sub(...) sub(x, y) returns x - y. The type of the result is based on the types of the arguments.

tan(...) tan(x) returns the tangent of x. x is measured in radians.

tanh(...) tanh(x) returns the hyperbolic tangent of x.

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

y0(...) y0(x) returns the Bessel function of the second kind of order 0 of x.

y1(...) y1(x) returns the Bessel function of the second kind of order 1 of x.

yn(...) yn(x,n) returns the Bessel function of the second kind of order n of x.

zero(...) zero(n) returns an *mpfr* inialized to 0.0 with the same sign as n. If n is not given, +0.0 is returned.

zeta(...) zeta(x) returns the Riemann zeta of x.

5.8 mpfr Formatting

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

__format__(...) x.__format__(fmt) returns 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 scientific format

Note: The formatting codes must be specified in the order shown above.

```
>>> import gmpy2
>>> 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'
```

CHAPTER

SIX

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 specifiy different precision and rounding modes for both the real and imaginary components of an *mpc*.

```
>>> import gmpy2
>>> from gmpy2 import mpc
>>> gmpy2.sqrt(mpc("1+2j"))
mpc('1.272019649514069+0.786151377757423 28j')
>>> gmpy2.get_context(real_prec=100,imag_prec=200)
context(precision=53, real_prec=100, imag_prec=200,
        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,
        trap_expbound=False,
        allow_complex=False)
>>> gmpy2.sqrt(mpc("1+2j"))
mpc('1.27201964951406896425242246173 76+0.786151377757423286069558585842958929523122057837723237664902
```

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)
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=True,
        trap_erange=False, erange=False,
        trap_divzero=False, divzero=False,
        trap_expbound=False,
```

```
        allow_complex=True)
>>> gmpy2.sqrt(-4)
mpc('0.0+2.0j')
```

6.1 mpc Methods

conjugate() Returns the complex conjugate.

digits() Returns a two element tuple where each element represents the real and imaginary components as a 3-tuple containing the mantissa, the exponent, and the number of bits of precision. The mantissa is represented as a string in the specified base with up to 'prec' digits. 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.

6.2 mpc Attributes

imag Returns the imaginary component.

precision Returns a 2-tuple containing the the precision of the real and imaginary components.

rc Returns a 2-tuple containing the ternary value of the real and imaginary components. The ternary value is 0 if the value of the component is exactly equal to the exact, infinite precision value. If the result code is 1, then the value of the component is greater than the exact value. If the result code is -1, then the value of the component is less than the exact, infinite precision value.

real Returns the real component.

6.3 mpc Functions

acos(...) acos(x) returns the arc-cosine of x.

acosh(...) acosh(x) returns the inverse hyperbolic cosine of x.

add(...) add(x, y) returns x + y. The type of the result is based on the types of the arguments.

asin(...) asin(x) returns the arc-sine of x.

asinh(...) asinh(x) return the inverse hyperbolic sine of x.

atan(...) atan(x) returns the arc-tangent of x.

atanh(...) atanh(x) returns the inverse hyperbolic tangent of x.

cos(...) cos(x) seturns the cosine of x.

cosh(...) cosh(x) returns the hyperbolic cosine of x.

div(...) div(x, y) returns x / y. The type of the result is based on the types of the arguments.

div_2exp(...) div_2exp(x, n) returns an 'mpfr' or 'mpc' divided by 2**n.

exp(...) exp(x) returns e**x.

fma(...) fma(x, y, z) returns correctly rounded result of (x * y) + z.

fms(...) fms(x, y, z) returns correctly rounded result of (x * y) - z.

is_inf(...) is_inf(x) returns True if either the real or imaginary component of x is Infinity or -Infinity.

is_nan(...) is_nan(x) returns True if either the real or imaginary component of x is NaN (Not-A-Number).

is_zero(...) is_zero(x) returns True if x is zero.

log(...) log(x) returns the natural logarithm of x.

log10(...) log10(x) returns the base-10 logarithm of x.

mpc(...) mpc(c[, precision=0]) returns a new 'mpc' object from an existing complex number (either a Python complex object or another 'mpc' object). If the precision is not specified, then the precision is taken from the current context. The rounding mode is always taken from the current context.

mpc(r[, i=0[, precision=0]]) returns a new 'mpc' object by converting two non-complex numbers into the real and imaginary components of an 'mpc' object. If the precision is not specified, then the precision is taken from the current context. The rounding mode is always taken from the current context.

mpc(s[, [precision=0[, base=10]]) returns a new 'mpc' object by converting a string s into a complex number. If base is omitted, then a base-10 representation is assumed otherwise a base between 2 and 36 can be specified. If the precision is not specified, then the precision is taken from the current context. The rounding mode is always taken from the current context.

In addition to the standard Python string representation of a complex number: `"1+2j"`, the string representation used by the MPC library: `"(1 2)"` is also supported.

Note: The precision can be specified either a single number that is used for both the real and imaginary components, or as a 2-tuple that can specify different precisions for the real and imaginary components.

---

mpc_random(...) mpfc_random(random_state) returns a uniformly distributed number in the unit square [0,1]x[0,1]. The parameter *random_state* must be created by random_state() first.

mul(...) mul(x, y) returns x * y. The type of the result is based on the types of the arguments.

mul_2exp(...) mul_2exp(x, n) returns 'mpfr' or 'mpc' multiplied by 2**n.

norm(...) norm(x) returns 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(...) phase(x) returns the phase angle, also known as argument, of a complex x.

polar(...) polar(x) returns the polar coordinate form of a complex x that is in rectangular form.

proj(...) proj(x) returns the projection of a complex x on to the Riemann sphere.

rect(...) rect(x) returns the polar coordinate form of a complex x that is in rectangular form.

sin(...) sin(x) returns the sine of x.

sinh(...) sinh(x) returns the hyberbolic sine of x.

sqrt(...) sqrt(x) returns the square root of x. If x is integer, rational, or real, then an *mpfr* will be returned. If x is complex, then an *mpc* will be returned. If context.allow_complex is True, negative values of x will return an *mpc*.

square(...) square(x) returns x * x. The type of the result is based on the types of the arguments.

sub(...) sub(x, y) returns x - y. The type of the result is based on the types of the arguments.

tan(...) tan(x) returns the tangent of x. x is measured in radians.

tanh(...) tanh(x) returns the hyperbolic tangent of x.

6.4 mpc Formatting

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

__format__(...) x.__format__(fmt) returns 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

Note: The formatting codes must be specified in the order shown above.

```
>>> import gmpy2
>>> from gmpy2 import mpc
>>> a=gmpy2.sqrt(mpc("1+2j"))
>>> a
mpc('1.272019649514069+0.786151377757423 28j')
>>> "{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   '
```

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*