bigfloat Documentation

Release 0.5.0-dev

Mark Dickinson

Dec 31, 2020
## Contents

1 Features 3

2 Introduction 5

3 Installation 7
   3.1 Where to get it 7
   3.2 Prerequisites 7
   3.3 Installation from the Python Package Index 7
   3.4 Platform-specific installation instructions 8
   3.5 Installation from source 8

4 Detailed Documentation 11
   4.1 Tutorial 11
   4.2 API Reference 17

5 Indices and tables 35

Python Module Index 37

Index 39
Release v0.5.0-dev.

The `bigfloat` package is a Python wrapper for the GNU MPFR library for arbitrary-precision floating-point reliable arithmetic. The MPFR library is a well-known portable C library for arbitrary-precision arithmetic on floating-point numbers. It provides precise control over precisions and rounding modes and gives correctly-rounded reproducible platform-independent results.

The `bigfloat` package aims to provide a convenient and friendly Python interface to the operations and functions provided by the MPFR library. The main class, `BigFloat`, gives an immutable multiple-precision floating-point type that can be freely mixed with Python integers and floats. The `Context` class, when used in conjunction with Python's `with` statement, gives a simple way of controlling precisions and rounding modes. Additional module-level functions provide various standard mathematical operations. There is full support for IEEE 754 signed zeros, nans, infinities and subnormals.
Features

• Supports Python 2 (version 2.7) and Python 3 (version 3.5 or later).
• Exactly reproducible correctly-rounded results across platforms; precisely-defined semantics compatible with the IEEE 754-2008 standard.
• Support for mixed-type operations with Python integers and floats.
• Support for emulating IEEE 754 arithmetic in any of the IEEE binary interchange formats described in IEEE 754-2008. Infinities, NaNs, signed zeros, and subnormals are all supported.
• Easy control of rounding modes and precisions via Context objects and Python’s with statement.
Introduction

Here's a quick tour:

```python
>>> from bigfloat import *
>>> sqrt(2, precision(100))  # compute sqrt(2) with 100 bits of precision
BigFloat.exact('1.4142135623730950488016887242092', precision=100)
>>> with precision(100):  # another way to get the same result
...    sqrt(2)
...
BigFloat.exact('1.4142135623730950488016887242092', precision=100)
>>> my_context = precision(100) + RoundTowardPositive
>>> my_context
Context(precision=100, rounding='RoundTowardPositive')
>>> sqrt(2, my_context)  # another, this time rounding up
BigFloat.exact('1.4142135623730950488016887242108', precision=100)
>>> with RoundTowardNegative:  # a lower bound for zeta(2)
...    sum(1/sqr(n) for n in range(1, 10000))
...
BigFloat.exact('1.6448340618469506', precision=53)
>>> zeta(2)  # actual value, for comparison
BigFloat.exact('1.6449340668482264', precision=53)
>>> const_pi()**2/6.0  # double check value
BigFloat.exact('1.6449340668482264', precision=53)
>>> quadruple_precision  # context implementing IEEE 754 binary128 format
Context(precision=113, emax=16384, emin=-16493, subnormalize=True)
>>> next_up(0, quadruple_precision)  # smallest subnormal for binary128
BigFloat.exact('-6.47517511943802511092443895822764655e-4966', precision=113)
>>> log2(_)  # double check value
BigFloat.exact('-16494.000000000000', precision=53)
```
CHAPTER 3

Installation

3.1 Where to get it

The latest released version of the *bigfloat* package can be obtained from the Python Package Index. Development sources can be checked out from the project’s GitHub page.

3.2 Prerequisites

The *bigfloat* package works with Python 2 (version 2.7) or Python 3 (version 3.5 or later). It uses a single codebase for both Python dialects, so the same source works on both dialects of Python.

Whether installing *bigfloat* from source or from the Python Package Index, you will need to have both the GMP and MPFR libraries already installed on your system, along with the include files for those libraries. See the MPFR homepage and the GMP homepage for more information about these libraries. Currently, MPFR version 3.0.0 or later is required.

On Ubuntu, prerequisites can be installed with:

```
sudo apt-get install libmpfr-dev
```

On Fedora Linux, use (for example):

```
su -c "yum install mpfr-devel"
```

On other flavours of Linux, some variant of one of the above should work.

3.3 Installation from the Python Package Index

Once you have the prerequisites described above installed, the recommended method for installation is to use *pip*:

```
If you prefer, you can use the `easy_install` command from `setuptools`:

```bash
easy-install bigfloat
```

Depending on your system, you may need superuser privileges for the install. For example:

```bash
sudo pip install bigfloat
```

If the MPFR and GMP libraries are installed in an unusual location, you may need to set appropriate environment variables when installing. For example, on an OS X 10.9 system with MPFR and GMP installed in `/opt/local`, I need to do:

```bash
sudo LIBRARY_PATH=/opt/local/lib CPATH=/opt/local/include pip install bigfloat
```

### 3.4 Platform-specific installation instructions

On a newly-installed version of Ubuntu 14.04LTS (trusty), the commands below are enough to install bigfloat for Python 3, from scratch. You may not need the first line if you already have `pip` and the Python development headers installed.

```bash
sudo apt-get install python3-dev python3-pip
sudo apt-get install libmpfr-dev
sudo pip3 install bigfloat
```

For Python 2, the procedure is similar:

```bash
sudo apt-get install python-dev python-pip
sudo apt-get install libmpfr-dev
sudo pip install bigfloat
```

On Fedora 20, the following sequence of commands worked for me (for Python 3; again, remove all occurrences of `3` to get the commands for Python 2):

```bash
su -c "yum install gcc python3-devel python3-pip"
su -c "yum install mpfr-devel"
su -c "pip-python3 install bigfloat"
```

### 3.5 Installation from source

Installation from source (for example, from a GitHub checkout, or from an unpacked source distribution), is similar to installation from the Python Package Index: the only difference is that you should use the `setup.py` script instead of using `pip` or `easy_install`. On many systems, installation should be as simple as typing:

```bash
python setup.py install
```

in the top-level directory of the unpacked distribution. As above, you may need superuser privileges to install the library, for example with:

```bash
sudo python setup.py install
```
Again as above, if the libraries and include files are installed in an unusual place, it may be necessary to specify their location using environment variables on the command line. For example:

```
LIBRARY_PATH=/opt/local/lib CPATH=/opt/local/include python setup.py install
```
4.1 Tutorial

Start by importing the contents of the package with:

```python
>>> from bigfloat import *
```

You should be a little bit careful here: this import brings a fairly large number of functions into the current namespace, some of which shadow existing Python builtins, namely `abs`, `max`, `min`, `pow`, `round`, and (on Python 2 only) `cmp`. In normal usage you’ll probably only want to import the classes and functions that you actually need.

4.1.1 BigFloat construction

The main type of interest is the BigFloat class. The BigFloat type is an immutable binary floating-point type. A BigFloat instance can be created from an integer, a float or a string:

```python
>>> BigFloat(123)
BigFloat.exact('123.000000000000000000000000000000000000000', precision=113)
>>> BigFloat('-4.56')
BigFloat.exact('-4.5599999999999999999999999999999966', precision=113)
```

Each BigFloat instance has both a value and a precision. The precision gives the number of bits used to store the significand of the BigFloat. The value of a finite nonzero BigFloat with precision p is a real number of the form 
\[-1\] * s * m * 2**e, where the sign s is either 0 or 1, the significand m is a number in the half-open interval (0.5, 1.0) that can be expressed in the form n/2**p for some integer n, and e is an integer giving the exponent. In addition, zeros (positive and negative), infinities and NaNs are representable. Just like Python floats, the printed form of a BigFloat shows only a decimal approximation to the exact stored value, for the benefit of human readers.

The precision of a newly-constructed BigFloat instance is dictated by the current precision, which defaults to 113 (the precision of the IEEE 754 “binary128” format, a.k.a. quadruple precision). This setting can be overridden by supplying the context keyword argument to the constructor:
The first argument to the `BigFloat` constructor is rounded to the correct precision using the current rounding mode, which defaults to `RoundTiesToEven`; again, this can be overridden with the `context` keyword argument:

```python
>>> BigFloat('3.14', context=RoundTowardZero)
BigFloat.exact('3.13999999999999999999999999999999972', precision=113)
```

More generally, the second argument to the `BigFloat` constructor can be any instance of the `Context` class. The various rounding modes are all `Context` instances, and `precision` is a function returning a `Context`:

```python
>>> RoundTowardNegative
Context(rounding=ROUND_TOWARD_NEGATIVE)
```

`Context` instances can be combined by addition, as seen above:

```python
>>> precision(1000) + RoundTowardNegative
Context(precision=1000, rounding=ROUND_TOWARD_NEGATIVE)
```

When adding two contexts that both specify values for a particular attribute, the value for the right-hand addend takes precedence:

```python
>>> c = Context(subnormalize=False, rounding=ROUND_TOWARD_POSITIVE)
>>> double_precision
Context(precision=53, emax=1024, emin=-1073, subnormalize=True)
>>> double_precision + c
Context(precision=53, emax=1024, emin=-1073, subnormalize=False, rounding=ROUND_TOWARD_POSITIVE)
```

The `bigfloat` package also defines various constant `Context` instances. For example, `quadruple_precision` is a `Context` object that corresponds to the IEEE 754 binary128 interchange format:

```python
>>> quadruple_precision
Context(precision=113, emax=16384, emin=-16493, subnormalize=True)
>>> BigFloat('1.1', quadruple_precision)
BigFloat.exact('1.10000000000000000000000000000000008', precision=113)
```

The current settings for precision and rounding mode are given by the `current context`, accessible via the `getcontext` function:

```python
>>> getcontext()
Context(precision=113, emax=16384, emin=-16493, subnormalize=True, rounding=ROUND_TIES_TO_EVEN)
```

There's also a `setcontext` function for changing the current context; however, the preferred method for making temporary changes to the current context is to use Python's `with` statement. More on this below.
Note that (in contrast to Python’s standard library `decimal` module), `Context` instances are immutable.

There’s a second method for constructing `BigFloat` instances: `BigFloat.exact`. Just like the usual constructor, `BigFloat.exact` accepts integers, floats and strings. However, for integers and floats it performs an exact conversion, creating a `BigFloat` instance with precision large enough to hold the integer or float exactly (regardless of the current precision setting):

```python
>>> BigFloat.exact(-123)
BigFloat.exact('123.0', precision=7)
>>> BigFloat.exact(7**30)
BigFloat.exact('22539340290692258087863249.0', precision=85)
>>> BigFloat.exact(-56.7)
BigFloat.exact('-56.700000000000003', precision=53)
```

For strings, `BigFloat.exact` accepts a second `precision` argument, and always rounds using the `ROUND_TIES_TO_EVEN` rounding mode.

```python
>>> BigFloat.exact('1.1', precision=80)
BigFloat.exact('1.1000000000000000000000003', precision=80)
```

The result of a call to `BigFloat.exact` is independent of the current context; this is why the `repr` of a `BigFloat` is expressed in terms of `BigFloat.exact`. The `str` of a `BigFloat` looks prettier, but doesn’t supply enough information to recover that `BigFloat` exactly if you don’t know the precision:

```python
>>> print(BigFloat('1e1000', precision(20)))
9.9999988e+999
>>> print(BigFloat('1e1000', precision(21)))
9.9999988e+999
```

### 4.1.2 Arithmetic on `BigFloat` instances

All the usual arithmetic operations apply to `BigFloat` instances, and those instances can be freely mixed with integers and floats (but not strings!) in those operations:

```python
>>> BigFloat(1234)/3
BigFloat.exact('411.333333333333333333333333333333317', precision=113)
>>> BigFloat('1e123')**0.5
BigFloat.exact('3.162277660168379319988935443271851e+616', precision=113)
```

As with the `BigFloat` constructor, the precision for the result is taken from the current context, as is the rounding mode used to round the exact mathematical result to the nearest `BigFloat`.

For mixed-type operations, the integer or float is converted *exactly* to a `BigFloat` before the operation (as though the `BigFloat.exact` constructor had been applied to it). So there’s only a single point where precision might be lost: namely, when the result of the operation is rounded to the nearest value representable as a `BigFloat`.

**Note:** The current precision and rounding mode even apply to the unary plus and minus operations. In particular, `+x` is not necessarily a no-op for a `BigFloat` instance `x`:

```python
>>> BigFloat.exact(7**100)
BigFloat.exact('323447650962475799134464776910021681085720319890462540093389
533139169145963692806001.0', precision=281)
>>> +BigFloat.exact(7**100)
BigFloat.exact('3.23447650962475799134464776910021692e+84', precision=113)
```
This makes the unary plus operator useful as a way to round a result produced in a different context to the current context.

For each arithmetic operation the bigfloat package exports a corresponding function. For example, the `div` function corresponds to usual (true) division:

```python
>>> 355/BigFloat(113)
BigFloat.exact('3.14159292035398230088495575221238935', precision=113)
>>> div(355, 113)
BigFloat.exact('3.14159292035398230088495575221238935', precision=113)
```

This is useful for a couple of reasons: one reason is that it makes it possible to use `div(x, y)` in contexts where a `BigFloat` result is desired but where one or both of `x` and `y` might be an integer or float. But a more important reason is that these functions, like the `BigFloat` constructor, accept an extra `context` keyword argument giving a context for the operation:

```python
>>> div(355, 113, context=single_precision)
BigFloat.exact('3.14159298', precision=24)
```

Similarly, the `sub` function corresponds to Python’s subtraction operation. To fully appreciate some of the subtleties of the ways that binary arithmetic operations might be performed, note the difference in the results of the following:

```python
>>> x = 10**16+1  # integer, not exactly representable as a float
>>> y = 10**16.   # 10.**16 is exactly representable as a float
>>> x - y
0.0
>>> BigFloat(x, double_precision) - BigFloat(y, double_precision)
BigFloat.exact('0', precision=53)
>>> sub(x, y, double_precision)
BigFloat.exact('1.0000000000000000', precision=53)
```

For the first subtraction, the integer is first converted to a float, losing accuracy, and then the subtraction is performed, giving a result of 0.0. The second case is similar: `x` and `y` are both explicitly converted to `BigFloat` instances, and the conversion of `x` again loses precision. In the third case, `x` and `y` are implicitly converted to `BigFloat` instances, and that conversion is exact, so the subtraction produces exactly the right answer.

Comparisons between `BigFloat` instances and integers or floats also behave as you’d expect them to; for these, there’s no need for a corresponding function.

### 4.1.3 Mathematical functions

The bigfloat package provides a number of standard mathematical functions. These functions follow the same rules as the arithmetic operations above:

- the arguments can be integers, floats or `BigFloat` instances
- integers and float arguments are converted exactly to `BigFloat` instances before the function is applied
- the result is a `BigFloat` instance, with the precision of the result, and the rounding mode used to obtain the result, taken from the current context.
- attributes of the current context can be overridden by providing an additional `context` keyword argument.

Here are some examples:
For a full list of the supported functions, see the Standard functions section of the API Reference.

4.1.4 Controlling the precision and rounding mode

We've seen one way of controlling precision and rounding mode, via the context keyword argument. There's another way that's often more convenient, especially when a single context change is supposed to apply to multiple operations: contexts can be used directly in Python's with statement.

For example, here we compute high-precision upper and lower-bounds for the thousandth harmonic number:

```python
>>> with precision(100):
...     with RoundTowardNegative:  # lower bound
...         lower_bound = sum(div(1, n) for n in range(1, 1001))
...     with RoundTowardPositive:  # upper bound
...         upper_bound = sum(div(1, n) for n in range(1, 1001))
...
>>> lower_bound
BigFloat.exact('7.4854708605503449126565182015873', precision=100)
>>> upper_bound
BigFloat.exact('7.4854708605503449126565182077593', precision=100)
```

The effect of the with statement is to change the current context for the duration of the with block; when the block exits, the previous context is restored. With statements can be nested, as seen above. Let's double-check the above results using the asymptotic formula for the nth harmonic number:

```python
>>> n = 1000
>>> with precision(100):
...     approx = log(n) + const_euler() + div(1, 2*n) - 1/(12*sqr(n))
...     approx
BigFloat.exact('7.4854708605503365793271531207983', precision=100)
```

The error in this approximation should be approximately \(-1/(120*n^4)\). Let's check it:

```python
>>> error = approx - lower_bound
>>> error
BigFloat.exact('-8.33333333333333331e-15', precision=113)
>>> -1/(120*pow(n, 4))
BigFloat.exact('-8.3333333333333333391e-15', precision=113)
```

1 See http://mathworld.wolfram.com/HarmonicNumber.html
A more permanent change to the context can be effected using the `setcontext` function, which takes a single argument of type `Context`:

```python
>>> setcontext(precision(30))
>>> sqrt(2)
BigFloat.exact('1.4142135624', precision=30)
>>> setcontext(RoundTowardZero)
>>> sqrt(2)
BigFloat.exact('1.4142135605', precision=30)
```

An important point here is that in any place that a context is used, only the attributes specified by that context are changed. For example, the context `precision(30)` only has the `precision` attribute, so only that attribute is affected by the `setcontext` call; the other attributes are not changed. Similarly, the `setcontext(RoundTowardZero)` line above doesn’t affect the precision.

There’s a `DefaultContext` constant giving the default context, so you can always restore the original default context as follows:

```python
>>> setcontext(DefaultContext)
```

**Note:** If `setcontext` is used within a `with` statement, its effects only last for the duration of the block following the `with` statement.

### 4.1.5 Flags

The `bigfloat` package also provides five global flags: ‘Inexact’, ‘Overflow’, ‘ZeroDivision’, ‘Underflow’, and ‘NanFlag’, along with methods to set and test these flags:

```python
>>> set_flagstate(set())  # clear all flags
>>> get_flagstate()  
set()
>>> exp(10**100)
BigFloat.exact('inf', precision=113)
>>> get_flagstate()  
{'Overflow', 'Inexact'}
```

These flags show that overflow occurred, and that the given result (infinity) was inexact. The flags are sticky: none of the standard operations ever clears a flag:

```python
>>> sqrt(2)
BigFloat.exact('1.41421356237309504880168872420969798', precision=113)
>>> get_flagstate()  
{'Overflow', 'Inexact'}
>>> set_flagstate(set())  # clear all flags
>>> sqrt(2)
BigFloat.exact('1.41421356237309504880168872420969798', precision=113)
>>> get_flagstate()  
{'Inexact'}
```

The functions `clear_flag`, `set_flag` and `test_flag` allow clearing, setting and testing of individual flags. Support for these flags is preliminary, and the API may change in future versions.
4.2 API Reference

4.2.1 The BigFloat class

The BigFloat class implements multiple-precision binary floating-point numbers. Each BigFloat instance has both a value and a precision; the precision is an integer giving the number of significant bits used to store the value. A finite nonzero BigFloat instance with precision p can be thought of as a (sign, significand, exponent) triple (s, m, e), representing the value \((-1)^{**s} * m * 2^{**e}\), where m is a value in the range \([0.5, 1.0)\) stored with p bits of precision. Thus m is of the form \(n/2^{**p}\) for some integer n with \(2^{**(p-1)} \leq n < 2^{**p}\).

In addition to nonzero finite numbers, BigFloat instances can also represent positive and negative infinity, positive and negative zero, and NaNs.

BigFloat instances should be treated as immutable.

class bigfloat.BigFloat(value, context=None)

Construct a new BigFloat instance from an integer, string, float or another BigFloat instance, using the rounding-mode and output format (precision, exponent bounds and subnormalization) given by the current context. If the context keyword argument is given, its value should be a Context instance and its attributes override those of the current context.

value can be an integer, string, float, or another BigFloat instance. In all cases the given value is rounded to the format (determined by precision, exponent limits and subnormalization) given by the current context, using the rounding mode specified by the current context. The integer 0 is always converted to positive zero.

as_integer_ratio(self)

Return a pair \((n, d)\) of integers such that \(n\) and \(d\) are relatively prime, \(d\) is positive, and the value of self is exactly \(n/d\).

If self is an infinity or nan then ValueError is raised. Negative and positive zero are both converted to (0, 1).

copy_abs(self)

Return a copy of self with the sign bit unset.

In contrast to abs(self), self.copy_abs() makes no use of the context, and the result has the same precision as the original.

copy_neg(self)

Return a copy of self with the opposite sign bit.

In construct to neg(self), self.copy_neg() makes no use of the context, and the result has the same precision as the original.

exact(cls, value, precision=None)

A class method to construct a new BigFloat instance from an integer, string, float or another BigFloat instance, doing an exact conversion where possible. Unlike the usual BigFloat constructor, this alternative constructor makes no use of the current context and will not affect the current flags.

If value is an integer, float or BigFloat, then the precision keyword must not be given, and the conversion is exact. The resulting BigFloat has a precision sufficiently large to hold the converted value exactly. If value is a string, then the precision argument must be given. The string is converted using the given precision and the RoundTiesToEven rounding mode.

fromhex(cls, value, context=Context)

Class method that constructs a new BigFloat instance from a hexadecimal string. Rounds to the current context using the given precision. If the context keyword argument is given, its value should be a Context instance and its attributes override those of the current context.
hex(self)
    Return a hexadecimal representation of a BigFloat. The advantage of the hexadecimal representation is that it represents the value of the BigFloat exactly.

precision
    Precision of a BigFloat instance, in bits.

Special methods

The BigFloat type has a full complement of special methods. Here are some brief notes on those methods, indicating possible deviations from expected behaviour.

- The repr of a BigFloat instance \(x\) is independent of the current context, and has the property that \(\text{eval(repr}(x))\) recovers \(x\) exactly.

- The ‘+’, ‘-’, ‘*’, ‘/’, ‘%’ and ‘**’ binary operators are supported. The ‘/’ operator implements true division, regardless of whether \texttt{from __future__ import division} is in effect or not. The result of ‘%’ has the same sign as the second argument, so follows the existing Python semantics for ‘%’ on Python floats.

- For the above operators, mixed-type operations involving a BigFloat and an integer or float are permitted. These behave as though the non BigFloat operand is first converted to a BigFloat with no loss of accuracy.

- The ‘+’ and ‘-’ unary operators and built-in abs function are supported. Note that these all round to the current context; in particular, ‘+x’ is not necessarily equal to ‘x’ for a BigFloat instance \(x\).

- The six comparison operators ‘==’, ‘<=’, ‘<’, ‘!=’, ‘>’, ‘>=’ are supported. Comparisons involving NaNs always return False, except in the case of ‘!=’ where they always return True. Again, comparisons with integers or floats are permitted, with the integer or float being converted exactly before the comparison; the context does not affect the result of a comparison.

- Conversions to int and long always round towards zero; conversions to float always use the \texttt{ROUND_TIES_TO_EVEN} rounding mode. Conversion to bool returns False for a nonzero BigFloat and True otherwise. None of these conversions is affected by the current context.

- On Python 3, \texttt{round}, \texttt{math.floor}, \texttt{math.ceil} and \texttt{math.trunc} all behave as expected, returning the appropriate integer. They are unaffected by the current context. Note that Python 2 does not provide type-specific support for these four functions; the functions will all work on Python, but only by doing an implicit conversion to the built-in float type first.

- BigFloat instances are hashable. The hash function obeys the rule that objects that compare equal should hash equal; in particular, if \(x == n\) for some BigFloat instance \(x\) and some Python int or long \(n\) then \(\text{hash}(x) == \text{hash}(n)\), and similarly for floats.

- BigFloat instances support \texttt{str.format}-based formatting, as described in PEP 3101. The format specifier is much as described in the PEP, except that there’s additional support for hexadecimal and binary output, and for specification of a rounding mode. The general form of the format specifier looks like this:

\[
[\text{fill}][\text{align}][\text{sign}][#][0][\text{minimumwidth}][.\text{precision}][\text{rounding}][\text{type}]
\]

The type field is a single letter, which may be any of the following. The 'e', 'E', 'f', 'F', 'g', 'G' and 'A' types behave in the same way as for regular floats. Only the 'a', 'A' and 'b' formats are particular to BigFloat instances. The type may also be omitted, in which case \texttt{str}-style formatting is performed.
The optional `rounding` field consists of a single letter describing the rounding direction to be used when converting a `BigFloat` instance to a decimal value. The default is to use round-ties-to-even. Valid values for this field are described in the table below.

| 'U' | Round toward positive |
| 'D' | Round toward negative |
| 'Y' | Round away from zero |
| 'Z' | Round toward zero |
| 'N' | Round ties to even |

Examples:

```python
>>> from bigfloat import sqrt
>>> '{0:.6f}'.format(sqrt(2))
'1.414214'
>>> '{0:.6Df}'.format(sqrt(2))  # round down
'1.414213'
>>> '{0:^+20.6e}'.format(sqrt(2))
'...+1.414214e+00....'
>>> '{0:a}'.format(sqrt(2))
'0x1.6a09e667f3bcdp+0'
>>> '{0:b}'.format(sqrt(2))
'1.0110101000001001100001110001100010011100110001101101p+0'
```

### 4.2.2 The `Context` class

A `Context` object is a simple immutable object that packages together attributes describing a floating-point format, together with a rounding mode.

```python
class bigfloat.Context(precision=None, emin=None, emax=None, subnormalize=None, rounding=None)
```

Create a new `Context` object with the given attributes. Not all attributes need to be specified. Note that all attributes of the generated `Context` are read-only. Attributes that are unset for this `Context` instance return `None`.

- **precision**
  - Precision of the floating-point format, given in bits. This should be an integer in the range `[PRECISION_MIN, PRECISION_MAX]`.

- **emax**
  - Maximum exponent allowed for this format. The largest finite number representable in the context self is 
    $$(1 - 2^{self.\text{precision}}) \times 2^{self.\text{emax}}.$$
**emin**

Minimum exponent allowed for this format. The smallest positive number representable in the context self is $0.5 \times 2^{\text{self.emin}}$.

**Note:** There’s nothing to stop you defining a context with emin > emax, but don’t expect to get sensible results if you do this.

**subnormalize**

A boolean value: True if the format has gradual underflow, and False otherwise. With gradual underflow, all finite floating-point numbers have a value that’s an integer multiple of $2^{(\text{emin}-1)}$.

**rounding**

The rounding mode of this Context, for example ROUND_TIES_TO_EVEN. The available rounding modes are described in the Rounding modes section. Note that the values RoundTiesToEven, etc. exported by the bigfloat package are Context instances, not rounding modes, so cannot be used directly here.

Context instances can be added. If x and y are Context instances then x + y is the Context whose attributes combine those of x and y. In the case that both x and y have a particular attribute set, the value for y takes precedence:

```python
>>> x = Context(precision=200, rounding=ROUND_TIES_TO_EVEN)
>>> y = Context(precision=53, subnormalize=True)
>>> x + y
Context(precision=53, subnormalize=True, rounding=ROUND_TIES_TO_EVEN)
>>> y + x
Context(precision=200, subnormalize=True, rounding=ROUND_TIES_TO_EVEN)
```

Context instances can be used in with statements to alter the current context. In effect,

```python
with c:
    <block>
```

behaves roughly like

```python
old_context = getcontext()
setcontext(c)
<block>
setcontext(old_context)
```

except that nesting of with statements works as you’d expect, and the old context is guaranteed to be restored even if an exception occurs during execution of the block.

Note that for Context instances x and y,

```python
with x + y:
    <block>
```

is exactly equivalent to

```python
with x:
    with y:
        <block>
```

The bigfloat package defines a number of predefined Context instances.

**bigfloat.DefaultContext**

The context that’s in use when the bigfloat package is first imported. It has precision of 53, large exponent bounds, no subnormalization, and the RoundTiesToEven rounding mode.
bigfloat.**EmptyContext**

Equal to `Context()`. Occasionally useful where a context is syntactically required for a with statement, but no change to the current context is desired. For example:

```python
if <want_extra_precision>:
    c = extra_precision(10)
else:
    c = EmptyContext
with c:
    <do calculation>
```

bigfloat.**half_precision**

bigfloat.**single_precision**

bigfloat.**double_precision**

bigfloat.**quadruple_precision**

These `Context` instances correspond to the binary16, binary32, binary64 and binary128 interchange formats described in IEEE 754-2008 (section 3.6). They’re all special cases of the `IEEEContext` function.

bigfloat.**IEEEContext**(bitwidth)

Return IEEE 754-2008 context for a given bit width.

The IEEE 754 standard specifies binary interchange formats with bitwidths 16, 32, 64, 128, and all multiples of 32 greater than 128. This function returns the context corresponding to the interchange format for the given bitwidth.

See section 3.6 of IEEE 754-2008 or the bigfloat source for more details.

bigfloat.**precision**(prec)

Return context specifying the given precision.

`precision(prec)` is exactly equivalent to `Context(precision=prec)`.

bigfloat.**rounding**(rnd)

Return a context giving the specified rounding mode.

`rounding(rnd)` is exactly equivalent to `Context(rounding=rnd)`.

bigfloat.**RoundTiesToEven**

bigfloat.**RoundTowardZero**

bigfloat.**RoundAwayFromZero**

bigfloat.**RoundTowardPositive**

bigfloat.**RoundTowardNegative**

Context objects corresponding to the five available rounding modes. `RoundTiesToEven` rounds the result of an operation or function to the nearest representable `BigFloat`, with ties rounded to the `BigFloat` whose least significant bit is zero. `RoundTowardZero` rounds results towards zero. `RoundAwayFromZero` rounds results away from zero. `RoundTowardPositive` rounds results towards positive infinity, and `RoundTowardNegative` rounds results towards negative infinity.

**Constants**

bigfloat.**PRECISION_MIN**

4.2. **API Reference**
bigfloat.PRECISION_MAX
Minimum and maximum precision that’s valid for Context and BigFloat instances. In the current implementation, PRECISION_MIN is 2 and PRECISION_MAX is $2^{31}-1$.

bigfloat.EMIN_MIN

bigfloat.EMIN_MAX
Minimum and maximum allowed values for the Context emin attribute. In the current implementation, EMIN_MIN == -EMIN_MAX == $1-2^{30}$.

bigfloat.EMAX_MIN

bigfloat.EMAX_MAX
Minimum and maximum allowed values for the Context emax attribute. In the current implementation, -EMAX_MIN == EMAX_MAX == $2^{30}-1$.

The current context

There can be many Context objects in existence at one time, but there’s only ever one current context. The current context is given by a thread-local Context instance. Whenever the BigFloat constructor is called, or any arithmetic operation or standard function computation is performed, the current context is consulted to determine:

- The format that the result of the operation or function should take (as specified by the precision, emax, emin and subnormalize attributes of the context), and
- The rounding mode to use when computing the result, as specified by the rounding attribute of the current context.

If an additional context keyword argument is given to the operation, function or constructor, then attributes from the context override the corresponding attributes in the current context. For example,

```python
sqrt(x, context=my_context)
```

is equivalent to

```python
with my_context:
    sqrt(x)
```

The current context can be read and written directly using the getcontext and setcontext functions.

bigfloat.getcontext()
Return the current context.
Also initialises the context the first time it’s called in each thread.

bigfloat.setcontext(context)
Set the current context to that given.
Attributes provided by context override those in the current context. If context doesn’t specify a particular attribute, the attribute from the current context shows through.

It’s usually neater to make a temporary change to the context using a with statement, as described above. There’s also one convenience function that’s often useful in calculations:

bigfloat.extra_precision(prec)
Return copy of the current context with the precision increased by prec. Equivalent to Context(precision=getcontext().precision + p).
```python
>>> getcontext().precision
53
>>> extra_precision(10).precision
63
>>> with extra_precision(20):
... gamma(1.5)
...
BigFloat.exact('0.88622692545275801364912', precision=73)
```

### 4.2.3 Rounding modes

**bigfloat.ROUND_TIES_TO_EVEN**

This is the default rounding mode. The number to be rounded is mapped to the nearest representable value. In the case where that number is exactly midway between the two closest representable values, it is mapped to the value with least significant bit set to zero.

**bigfloat.ROUND_TOWARD_ZERO**

The number to be rounded is mapped to the nearest representable value that’s smaller than or equal to the original number in absolute value.

**bigfloat.ROUND_AWAY_FROM_ZERO**

The number to be rounded is mapped to the nearest representable value that’s greater than or equal to the original number in absolute value.

**bigfloat.ROUND_TOWARD_POSITIVE**

The number to be rounded is mapped to the nearest representable value greater than or equal to the original number.

**bigfloat.ROUND_TOWARD_NEGATIVE**

The number to be rounded is mapped to the nearest representable value less than or equal to the original number.

### 4.2.4 Standard functions

All functions in this section follow the same rules:

- Arguments can be `BigFloat` instances, integers or floats, unless otherwise specified.
- Integer or float arguments are converted exactly to `BigFloat` instances.
- The format of the result and the rounding mode used to obtain that result are taken from the current context.
- Attributes of the current context can be overridden by supplying an explicit `context` keyword argument.
- Results are correctly rounded.

#### Arithmetic functions

**bigfloat.add** *(x, y, context=None)*

Return \( x + y \).

**bigfloat.sub** *(x, y, context=None)*

Return \( x - y \).

**bigfloat.mul** *(x, y, context=None)*

Return \( x \times y \).
**bigfloat Documentation, Release 0.5.0-dev**

**bigfloat.div** \((x, y, context=None)\)

Return \(x\) divided by \(y\).

**bigfloat.pow** \((x, y, context=None)\)

Return \(x\) raised to the power \(y\).

Special values are handled as described in the ISO C99 and IEEE 754-2008 standards for the pow function.

- \(\text{pow}(\pm 0, y)\) returns plus or minus infinity for \(y\) a negative odd integer.
- \(\text{pow}(\pm 0, y)\) returns plus infinity for \(y\) negative and not an odd integer.
- \(\text{pow}(\pm 0, y)\) returns plus or minus zero for \(y\) a positive odd integer.
- \(\text{pow}(\pm 0, y)\) returns plus zero for \(y\) positive and not an odd integer.
- \(\text{pow}(-1, \pm \text{Inf})\) returns 1.
- \(\text{pow}(+1, y)\) returns 1 for any \(y\), even a NaN.
- \(\text{pow}(x, \pm 0)\) returns 1 for any \(x\), even a NaN.
- \(\text{pow}(x, y)\) returns NaN for finite negative \(x\) and finite non-integer \(y\).
- \(\text{pow}(x, -\text{Inf})\) returns plus infinity for \(0 < \text{abs}(x) < 1\), and plus zero for \(\text{abs}(x) > 1\).
- \(\text{pow}(x, +\text{Inf})\) returns plus zero for \(0 < \text{abs}(x) < 1\), and plus infinity for \(\text{abs}(x) > 1\).
- \(\text{pow}(-\text{Inf}, y)\) returns minus zero for \(y\) a negative odd integer.
- \(\text{pow}(-\text{Inf}, y)\) returns plus zero for \(y\) negative and not an odd integer.
- \(\text{pow}(-\text{Inf}, y)\) returns minus infinity for \(y\) a positive odd integer.
- \(\text{pow}(-\text{Inf}, y)\) returns plus infinity for \(y\) positive and not an odd integer.
- \(\text{pow}(+\text{Inf}, y)\) returns plus zero for \(y\) negative, and plus infinity for \(y\) positive.

**bigfloat.fmod** \((x, y, context=None)\)

Return \(x\) reduced modulo \(y\).

Returns the value of \(x - n \cdot y\), where \(n\) is the integer quotient of \(x\) divided by \(y\), rounded toward zero.

Special values are handled as described in Section F.9.7.1 of the ISO C99 standard: If \(x\) is infinite or \(y\) is zero, the result is NaN. If \(y\) is infinite and \(x\) is finite, the result is \(x\) rounded to the current context. If the result is zero, it has the sign of \(x\).

**bigfloat.floordiv** \((x, y, context=None)\)

Return the floor of \(x\) divided by \(y\).

The result is a BigFloat instance, rounded to the context if necessary. Special cases match those of the div function.

**bigfloat.mod** \((x, y, context=None)\)

Return the remainder of \(x\) divided by \(y\), with sign matching that of \(y\).

**bigfloat.remainder** \((x, y, context=None)\)

Return \(x\) reduced modulo \(y\).

Returns the value of \(x - n \cdot y\), where \(n\) is the integer quotient of \(x\) divided by \(y\), rounded to the nearest integer (ties rounded to even).

Special values are handled as described in Section F.9.7.1 of the ISO C99 standard: If \(x\) is infinite or \(y\) is zero, the result is NaN. If \(y\) is infinite and \(x\) is finite, the result is \(x\) (rounded to the current context). If the result is zero, it has the sign of \(x\).
**bigfloat Documentation, Release 0.5.0-dev**

**bigfloat.dim** *(x, y, context=None)*
Return \(\max(x - y, 0)\).
Return \(x - y\) if \(x > y\), +0 if \(x \leq y\), and NaN if either \(x\) or \(y\) is NaN.

**bigfloat.pos** *(x, context=None)*
Return \(x\).
As usual, the result is rounded to the current context. The \(\text{pos}\) function can be useful for rounding an intermediate result, computed with a temporary increase in precision, back to the current context. For example:

```python
>>> from bigfloat import pos, pow, precision
>>> pow(3, 60) + 1.234 - pow(3, 60)  # inaccurate due to precision loss
BigFloat.exact('1.23400115966796875000000000000000000', precision=113)
>>> with precision(200):  # compute result with extra precision
...     x = pow(3, 60) + 1.234 - pow(3, 60)
...     ...
... >>> pos(x)  # round back to original precision
BigFloat.exact('1.23399999999999998578914528479799628', precision=113)
```

**bigfloat.neg** *(x, context=None)*
Return \(-x\).

**bigfloat.abs** *(x, context=None)*
Return abs(x).

**bigfloat.fma** *(x, y, z, context=None)*
Return \((x \times y) + z\), with a single rounding according to the current context.

**bigfloat.fms** *(x, y, z, context=None)*
Return \((x \times y) - z\), with a single rounding according to the current context.

**bigfloat.sqr** *(x, context=None)*
Return the square of \(x\).

**bigfloat.sqrt** *(x, context=None)*
Return the square root of \(x\).
Return -0 if \(x\) is -0, to be consistent with the IEEE 754 standard. Return NaN if \(x\) is negative.

**bigfloat.rec_sqrt** *(x, context=None)*
Return the reciprocal square root of \(x\).
Return +Inf if \(x\) is \(\pm 0\), +0 if \(x\) is +Inf, and NaN if \(x\) is negative.

**bigfloat.cbrt** *(x, context=None)*
Return the cube root of \(x\).
For \(x\) negative, return a negative number. The cube root of -0 is defined to be -0.

**bigfloat.root** *(x, k, context=None)*
Return the \(k\)th root of \(x\).
For \(k\) odd and \(x\) negative (including -Inf), return a negative number. For \(k\) even and \(x\) negative (including -Inf), return NaN.
The \(k\)th root of -0 is defined to be -0, whatever the parity of \(k\).
This function is only implemented for nonnegative \(k\).

---

**4.2. API Reference**  25
bigfloat.hypot(x, y, context=None)
   Return the Euclidean norm of x and y, i.e., the square root of the sum of the squares of x and y.

Exponential and logarithmic functions

bigfloat.exp(x, context=None)
   Return the exponential of x.
bigfloat.exp2(x, context=None)
   Return two raised to the power x.
bigfloat.exp10(x, context=None)
   Return ten raised to the power x.
bigfloat.log(x, context=None)
   Return the natural logarithm of x.
bigfloat.log2(x, context=None)
   Return the base-two logarithm of x.
bigfloat.log10(x, context=None)
   Return the base-ten logarithm of x.
bigfloat.expm1(x, context=None)
   Return one less than the exponential of x.
bigfloat.log1p(x, context=None)
   Return the logarithm of one plus x.

Trigonometric functions

bigfloat.cos(x, context=None)
   Return the cosine of x.
bigfloat.sin(x, context=None)
   Return the sine of x.
bigfloat.tan(x, context=None)
   Return the tangent of x.
bigfloat.sec(x, context=None)
   Return the secant of x.
bigfloat.csc(x, context=None)
   Return the cosecant of x.
bigfloat.cot(x, context=None)
   Return the cotangent of x.

The above six trigonometric functions are inefficient for large arguments (for example, $x$ larger than BigFloat('1e1000000')), since reducing $x$ correctly modulo $\pi$ requires computing $\pi$ to high precision. Input arguments are in radians, not degrees.

bigfloat.acos(x, context=None)
   Return the inverse cosine of $x$.

   The mathematically exact result lies in the range $[0, \pi]$. However, note that as a result of rounding to the current context, it’s possible for the actual value returned to be fractionally larger than $\pi$: 
>>> from bigfloat import precision
>>> with precision(12):
...   x = acos(-1)
...   ...
>>> print(x)
3.1416
>>> x > const_pi()
True

bigfloat.asin(x, context=None)
Return the inverse sine of x.

The mathematically exact result lies in the range [-\(\pi/2, \pi/2\)]. However, note that as a result of rounding to the current context, it’s possible for the actual value to lie just outside this range.

bigfloat.atan(x, context=None)
Return the inverse tangent of x.

The mathematically exact result lies in the range [-\(\pi/2, \pi/2\)]. However, note that as a result of rounding to the current context, it’s possible for the actual value to lie just outside this range.

These functions return a result in radians.

bigfloat.atan2(y, x, context=None)
Return \(\text{atan}(y / x)\) with the appropriate choice of function branch.

If \(x > 0\), then \(\text{atan2}(y, x)\) is mathematically equivalent to \(\text{atan}(y / x)\). If \(x < 0\) and \(y > 0\), \(\text{atan}(y, x)\) is equivalent to \(\pi + \text{atan}(y, x)\). If \(x < 0\) and \(y < 0\), the result is \(-\pi + \text{atan}(y, x)\).

Geometrically, \(\text{atan2}(y, x)\) is the angle (measured counterclockwise, in radians) from the positive x-axis to the line segment joining (0, 0) to (x, y), in the usual representation of the x-y plane.

Special values are handled as described in the ISO C99 and IEEE 754-2008 standards for the atan2 function. The following examples illustrate the rules for positive y; for negative y, apply the symmetry \(\text{atan}(-y, x) = -\text{atan}(y, x)\).

>>> finite = positive = 2.3
>>> negative = -2.3
>>> inf = BigFloat('inf')

>>> print(atan2(+0.0, -0.0)) # pi
3.14159265358979323846264338327950280
>>> print(atan2(+0.0, +0.0)) # 0
0
>>> print(atan2(+0.0, negative)) # pi
3.14159265358979323846264338327950280
>>> print(atan2(+0.0, positive)) # 0
0
>>> print(atan2(positive, 0.0)) # pi / 2
1.570796326794896619231269163975140
>>> print(atan2(inf, -inf)) # 3*pi / 4
2.35619449019234492884698253745962710
>>> print(atan2(inf, inf)) # pi / 4
0.785398163397448309615660845819875699
>>> print(atan2(inf, finite)) # pi / 2
1.570796326794896619231269163975140
>>> print(atan2(positive, -inf)) # pi
3.14159265358979323846264338327950280

(continues on next page)
Hyperbolic trig functions

```python
>>> print(atan2(positive, +inf))  # 0
0
```

bigfloat.<code>cosh</code>(x, context=None)
Return the hyperbolic cosine of x.

bigfloat.<code>sinh</code>(x, context=None)
Return the hyperbolic sine of x.

bigfloat.<code>tanh</code>(x, context=None)
Return the hyperbolic tangent of x.

bigfloat.<code>sech</code>(x, context=None)
Return the hyperbolic secant of x.

bigfloat.<code>csch</code>(x, context=None)
Return the hyperbolic cosecant of x.

bigfloat.<code>coth</code>(x, context=None)
Return the hyperbolic cotangent of x.

bigfloat.<code>acosh</code>(x, context=None)
Return the inverse hyperbolic cosine of x.

bigfloat.<code>asinh</code>(x, context=None)
Return the inverse hyperbolic sine of x.

bigfloat.<code>atanh</code>(x, context=None)
Return the inverse hyperbolic tangent of x.

Special functions

bigfloat.<code>eint</code>(x, context=None)
Return the exponential integral of x.

bigfloat.<code>li2</code>(x, context=None)
Return the real part of the dilogarithm of x.

bigfloat.<code>gamma</code>(x, context=None)
Return the value of the Gamma function of x.

bigfloat.<code>lngamma</code>(x, context=None)
Return the logarithm of the absolute value of the Gamma function at x.

bigfloat.<code>lgamma</code>(x, context=None)
Return the logarithm of the absolute value of the Gamma function at x.

bigfloat.<code>zeta</code>(x, context=None)
Return the value of the Riemann zeta function on x.

bigfloat.<code>zeta_ui</code>(x, context=None)
Return the value of the Riemann zeta function at the nonnegative integer x.

bigfloat.<code>erf</code>(x, context=None)
Return the value of the error function at x.
bigfloat.\texttt{erfc}(x, context=None)

Return the value of the complementary error function at $x$.

bigfloat.\texttt{j0}(x, context=None)

Return the value of the first kind Bessel function of order 0 at $x$.

bigfloat.\texttt{j1}(x, context=None)

Return the value of the first kind Bessel function of order 1 at $x$.

bigfloat.\texttt{jn}(n, x, context=None)

Return the value of the first kind Bessel function of order $n$ at $x$.

$n$ should be a Python integer.

bigfloat.\texttt{y0}(x, context=None)

Return the value of the second kind Bessel function of order 0 at $x$.

bigfloat.\texttt{y1}(x, context=None)

Return the value of the second kind Bessel function of order 1 at $x$.

bigfloat.\texttt{yn}(n, x, context=None)

Return the value of the second kind Bessel function of order $n$ at $x$.

$n$ should be a Python integer.

bigfloat.\texttt{agm}(x, y, context=None)

Return the arithmetic geometric mean of $x$ and $y$.

bigfloat.\texttt{factorial}(x, context=None)

Return the factorial of the nonnegative integer $x$.

\section*{Constants}

bigfloat.\texttt{const_catalan}(context=None)

Return Catalan's constant.

Returns the value of Catalan’s constant 0.9159655941\ldots, with precision and rounding mode taken from the current context. The Catalan constant is defined as the limit of the series $1 - 1/3^{2} + 1/5^{2} - 1/7^{2} + 1/9^{2} - \ldots$

bigfloat.\texttt{const_euler}(context=None)

Return Euler's constant.

Returns the value of the Euler-Mascheroni constant, 0.5772156649\ldots, with precision and rounding mode taken from the current context. The constant is equal to the limit of $(1 + 1/2 + 1/3 + \ldots + 1/n) - \log(n)$ as $n$ approaches infinity.

bigfloat.\texttt{const_log2}(context=None)

Return $\log(2)$.

Returns the natural logarithm of 2, 0.6931471805\ldots, with precision and rounding mode taken from the current context.

bigfloat.\texttt{const_pi}(context=None)

Return $\pi$.

Returns $\pi = 3.1415926535\ldots$, with precision and rounding mode taken from the current context.
### Miscellaneous functions

**bigfloat.max** 
\(x, y, context=None\)  
Return the maximum of \(x\) and \(y\).

If \(x\) and \(y\) are both NaN, return NaN. If exactly one of \(x\) and \(y\) is NaN, return the non-NaN value. If \(x\) and \(y\) are zeros of different signs, return +0.

**bigfloat.min** 
\(x, y, context=None\)  
Return the minimum of \(x\) and \(y\).

If \(x\) and \(y\) are both NaN, return NaN. If exactly one of \(x\) and \(y\) is NaN, return the non-NaN value. If \(x\) and \(y\) are zeros of different signs, return -0.

**bigfloat.copysign**  
\(x, y, context=None\)  
Return a new BigFloat object with the magnitude of \(x\) but the sign of \(y\).

**bigfloat.frac**  
\(x, context=None\)  
Return the fractional part of \(x\).  
The result has the same sign as \(x\).

**bigfloat.ceil**  
\(x, context=None\)  
Return the next higher or equal integer to \(x\).

If the result is not exactly representable, it will be rounded according to the current context. Note that the rounding step means that it’s possible for the result to be smaller than \(x\). For example:

```python
>>> x = 2**1000 + 1  
>>> ceil(2**1000 + 1) >= x
False
```

One way to be sure of getting a result that’s greater than or equal to \(x\) is to use the `RoundTowardPositive` rounding mode:

```python
>>> with RoundTowardPositive:
...     x = 2**100 + 1  
...     ceil(x) >= x
...     True
```

Similar comments apply to the `floor`, `round` and `trunc` functions.

**Note:** This function corresponds to the MPFR function `mpfr_rint ceil`, not to `mpfr.ceil`.

**bigfloat.floor**  
\(x, context=None\)  
Return the next lower or equal integer to \(x\).

If the result is not exactly representable, it will be rounded according to the current context.

Note that it’s possible for the result to be larger than \(x\). See the documentation of the `ceil` function for more information.

**Note:** This function corresponds to the MPFR function `mpfr_rint_floor`, not to `mpfr.floor`.

**bigfloat.round**  
\(x, context=None\)  
Return the nearest integer to \(x\), rounding halfway cases away from zero.
If the result is not exactly representable, it will be rounded according to the current context.

**Note:** This function corresponds to the MPFR function `mpfr_rint_round`, not to `mpfr_round`.

**bigfloat.trunc**(*x*, *context=None*)

Return the next integer towards zero.

If the result is not exactly representable, it will be rounded according to the current context.

**Note:** This function corresponds to the MPFR function `mpfr_rint_trunc`, not to `mpfr_trunc`.

### 4.2.5 Other Functions

These are the functions exported by the `bigfloat` package that don’t fit into the above section, for one reason or another.

**Comparisons**

These functions provide three-way comparisons.

**bigfloat.sgn**(*x*)

Return the sign of *x*.

- Return a positive integer if *x* > 0, 0 if *x* == 0, and a negative integer if *x* < 0. Raise `ValueError` if *x* is a NaN.
- This function is equivalent to `cmp(x, 0)`, but more efficient.

**bigfloat.cmp**(*op1*, *op2*)

Perform a three-way comparison of *op1* and *op2*.

- Return a positive value if *op1* > *op2*, zero if *op1* = *op2*, and a negative value if *op1* < *op2*. Both *op1* and *op2* are considered to their full own precision, which may differ. If one of the operands is NaN, raise `ValueError`.
- Note: This function may be useful to distinguish the three possible cases. If you need to distinguish two cases only, it is recommended to use the predicate functions like ‘greaterequal’; they behave like the IEEE 754 comparisons, in particular when one or both arguments are NaN.

**bigfloat.cmpabs**(*op1*, *op2*)

Compare the absolute values of *op1* and *op2*.

- Return a positive value if *op1* > *op2*, zero if *op1* = *op2*, and a negative value if *op1* < *op2*. Both *op1* and *op2* are considered to their full own precision, which may differ. If one of the operands is NaN, raise `ValueError`.
- Note: This function may be useful to distinguish the three possible cases. If you need to distinguish two cases only, it is recommended to use the predicate functions like ‘greaterequal’; they behave like the IEEE 754 comparisons, in particular when one or both arguments are NaN.

The following functions match the functionality of the builtin Python comparison operators.

**bigfloat.greater**(*x*, *y*)

Return True if *x* > *y* and False otherwise.

- This function returns False whenever *x* and/or *y* is a NaN.

**bigfloat.greaterequal**(*x*, *y*)

Return True if *x* >= *y* and False otherwise.
This function returns False whenever x and/or y is a NaN.

```python
bigfloat.less(x, y)
```
Return True if x < y and False otherwise.

This function returns False whenever x and/or y is a NaN.

```python
bigfloat.lesequal(x, y)
```
Return True if x <= y and False otherwise.

This function returns False whenever x and/or y is a NaN.

```python
bigfloat.equal(x, y)
```
Return True if x == y and False otherwise.

This function returns False whenever x and/or y is a NaN.

```python
bigfloat.equal(x, y)
```
Return True if x != y and False otherwise.

This function returns True whenever x and/or y is a NaN.

New in version 0.4: The `notequal` function was added in version 0.4.

There are two additional comparison functions that don’t correspond to any of the Python comparison operators.

```python
bigfloat.lessgreater(x, y)
```
Return True if x < y or x > y and False otherwise.

This function returns False whenever x and/or y is a NaN.

```python
bigfloat.unordered(x, y)
```
Return True if x or y is a NaN and False otherwise.

**Number classification functions**

The following functions all accept a single `BigFloat` instance (or a float, or an integer) and return a boolean value. They make no use of the current context, and do not affect the state of the flags.

```python
bigfloat.is_nan(x)
```
Return True if x is a NaN, else False.

```python
bigfloat.is_inf(x)
```
Return True if x is an infinity, else False.

```python
bigfloat.is_zero(x)
```
Return True if x is a zero, else False.

```python
bigfloat.is_finite(x)
```
Return True if x is finite, else False.

A BigFloat instance is considered to be finite if it is neither an infinity or a NaN.

```python
bigfloat.is_negative(x)
```
Return True if x has its sign bit set, else False.

Note that this function returns True for negative zeros.

```python
bigfloat.is_integer(x)
```
Return True if x is an exact integer, else False.
### Miscellaneous functions

**bigfloat.next_up**(x, context=None)

`next_up(x)`: return the least representable float that’s strictly greater than x.

This operation is quiet: flags are not affected.

**bigfloat.next_down**(x, context=None)

`next_down(x)`: return the greatest representable float that’s strictly less than x.

This operation is quiet: flags are not affected.

### 4.2.6 Flags

**bigfloat.Underflow**

Underflow flag. Set whenever the result of an operation underflows. The meaning of this flag differs depending on whether the subnormalize attribute is true for the operation context. In the language of IEEE 754, we use the after rounding semantics. The Underflow flag is set on underflow even when the result of an operation is exact.

In detail: let c be the context that’s in effect for an operation, function or BigFloat construction. Let x be the result of the operation, rounded to the context precision with the context rounding mode, but as though the exponent were unbounded.

If c.subnormalize is False, the Underflow flag is set if and only if x is nonzero, finite, and strictly smaller than $2^{*(c.emin-1)}$ in absolute value. If c.subnormalize is True, the Underflow flag is set if and only if x is nonzero, finite, and strictly smaller than $2^{*(c.emin+c.precision-2)}$ in absolute value.

**bigfloat.Overflow**

Set whenever the result of an operation overflows. An operation performed in a context c overflows if the result computed as if with unbounded exponent range is finite and greater than or equal to $2^{c.emax}$ in absolute value.

**bigfloat.ZeroDivision**

Set whenever an exact infinite result is obtained from finite inputs. Despite the name, this flag is not just set for divisions by zero. For example, log(0) will set the ZeroDivision flag.

**bigfloat.Inexact**

Inexact flag. Set whenever the result of an operation is not exactly equal to the true mathematical result.

**bigfloat.NanFlag**

NaN flag. Set whenever the result of an operation gives a NaN result.

**bigfloat.clear_flag**(f)

Clear the given flag.

**bigfloat.set_flag**(f)

Set the given flag.

**bigfloat.test_flag**(f)

Return True if the given flag is set, and False otherwise.

**bigfloat.get_flagstate()**

Return a set containing the flags that are currently set.

**bigfloat.set_flagstate**(flagset)

Set all flags in flagset, and clear all other flags.
4.2.7 MPFR Version information

**bigfloat**.\texttt{MPFR\_VERSION\_STRING}
The version of the MPFR library being used, as a string.

**bigfloat**.\texttt{MPFR\_VERSION}
The version of the MPFR library being used, as an integer.

**bigfloat**.\texttt{MPFR\_VERSION\_MAJOR}
An integer giving the major level of the MPFR version.

**bigfloat**.\texttt{MPFR\_VERSION\_MINOR}
An integer giving the minor level of the MPFR version.

**bigfloat**.\texttt{MPFR\_VERSION\_PATCHLEVEL}
An integer giving the patch level of the MPFR version.
CHAPTER 5

Indices and tables

• genindex
b

bigfloat, 17
## Index

| A | abs() (in module bigfloat), 25  
|  | acos() (in module bigfloat), 26  
|  | acosh() (in module bigfloat), 28  
|  | add() (in module bigfloat), 23  
|  | agm() (in module bigfloat), 29  
|  | as_integer_ratio() (bigfloat.BigFloat method), 17  
|  | asin() (in module bigfloat), 27  
|  | asinh() (in module bigfloat), 28  
|  | atan() (in module bigfloat), 27  
|  | atan2() (in module bigfloat), 27  
|  | atanh() (in module bigfloat), 28  
| B | BigFloat (class in bigfloat), 17  
|  | bigfloat (module), 1, 17  
| C | cbrt() (in module bigfloat), 25  
|  | ceil() (in module bigfloat), 30  
|  | clear_flag() (in module bigfloat), 33  
|  | cmp() (in module bigfloat), 31  
|  | cmpabs() (in module bigfloat), 31  
|  | const_catalan() (in module bigfloat), 29  
|  | const_euler() (in module bigfloat), 29  
|  | const_log2() (in module bigfloat), 29  
|  | const_pi() (in module bigfloat), 29  
|  | Context (class in bigfloat), 19  
|  | copy_abs() (bigfloat.BigFloat method), 17  
|  | copy_neg() (bigfloat.BigFloat method), 17  
|  | copysign() (in module bigfloat), 30  
|  | cos() (in module bigfloat), 26  
|  | cosh() (in module bigfloat), 28  
|  | cot() (in module bigfloat), 26  
|  | coth() (in module bigfloat), 28  
|  | csc() (in module bigfloat), 26  
|  | csch() (in module bigfloat), 28  
| D | DefaultContext (in module bigfloat), 20  
|  | dim() (in module bigfloat), 24  
|  | div() (in module bigfloat), 23  
|  | double_precision (in module bigfloat), 21  
| E | eint() (in module bigfloat), 28  
|  | emax (bigfloat.Context attribute), 19  
|  | EMAX_MAX (in module bigfloat), 22  
|  | EMAX_MIN (in module bigfloat), 22  
|  | emin (bigfloat.Context attribute), 19  
|  | EMIN_MAX (in module bigfloat), 22  
|  | EMIN_MIN (in module bigfloat), 22  
|  | EmptyContext (in module bigfloat), 21  
|  | equal() (in module bigfloat), 32  
|  | erf() (in module bigfloat), 28  
|  | erfc() (in module bigfloat), 28  
|  | exact() (bigfloat.BigFloat method), 17  
|  | exp() (in module bigfloat), 26  
|  | exp10() (in module bigfloat), 26  
|  | exp2() (in module bigfloat), 26  
|  | expm1() (in module bigfloat), 26  
|  | extra_precision() (in module bigfloat), 22  
| F | factorial() (in module bigfloat), 29  
|  | floor() (in module bigfloat), 30  
|  | floordiv() (in module bigfloat), 24  
|  | fma() (in module bigfloat), 25  
|  | fmod() (in module bigfloat), 24  
|  | fms() (in module bigfloat), 25  
|  | frac() (in module bigfloat), 30  
|  | fromhex() (bigfloat.BigFloat method), 17  
| G | gamma() (in module bigfloat), 28  
|  | get_flagstate() (in module bigfloat), 33  
|  | getcontext() (in module bigfloat), 22  

39
greater() (in module bigfloat), 31
greaterequal() (in module bigfloat), 31

H
half_precision (in module bigfloat), 21
hex() (bigfloat.BigFloat method), 17
hypot() (in module bigfloat), 25

I
IEEEContext() (in module bigfloat), 21
Inexact (in module bigfloat), 33
is_finite() (in module bigfloat), 32
is_inf() (in module bigfloat), 32
is_integer() (in module bigfloat), 32
is_nan() (in module bigfloat), 32
is_negative() (in module bigfloat), 32
is_zero() (in module bigfloat), 32

J
j0() (in module bigfloat), 29
j1() (in module bigfloat), 29
jn() (in module bigfloat), 29

L
less() (in module bigfloat), 32
lessequal() (in module bigfloat), 32
lessgreater() (in module bigfloat), 32
lgamma() (in module bigfloat), 28
li2() (in module bigfloat), 28
lngamma() (in module bigfloat), 28
log() (in module bigfloat), 26
log10() (in module bigfloat), 26
log1p() (in module bigfloat), 26
log2() (in module bigfloat), 26

M
max() (in module bigfloat), 30
min() (in module bigfloat), 30
mod() (in module bigfloat), 24
MPFR_VERSION (in module bigfloat), 34
MPFR_VERSION_MAJOR (in module bigfloat), 34
MPFR_VERSION_MINOR (in module bigfloat), 34
MPFR_VERSION_PATCHLEVEL (in module bigfloat), 34
MPFR_VERSION_STRING (in module bigfloat), 34
mul() (in module bigfloat), 23

N
NanFlag (in module bigfloat), 33
neg() (in module bigfloat), 25
next_down() (in module bigfloat), 33
next_up() (in module bigfloat), 33
notequal() (in module bigfloat), 32

O
Overflow (in module bigfloat), 33

P
pos() (in module bigfloat), 25
pow() (in module bigfloat), 24
precision (bigfloat.BigFloat attribute), 18
precision (bigfloat.Context attribute), 19
precision() (in module bigfloat), 21
PRECISION_MAX (in module bigfloat), 21
PRECISION_MIN (in module bigfloat), 21

Q
quadruple_precision (in module bigfloat), 21

R
rec_sqrt() (in module bigfloat), 25
remainder() (in module bigfloat), 24
root() (in module bigfloat), 25
round() (in module bigfloat), 30
ROUND_AWAY_FROM_ZERO (in module bigfloat), 23
ROUND_TIES_TO_EVEN (in module bigfloat), 23
ROUND_TOWARD_NEGATIVE (in module bigfloat), 23
ROUND_TOWARD_POSITIVE (in module bigfloat), 23
ROUND_TOWARD_ZERO (in module bigfloat), 23
RoundAwayFromZero (in module bigfloat), 21
rounding (bigfloat.Context attribute), 20
rounding() (in module bigfloat), 21
RoundTiesToEven (in module bigfloat), 21
RoundTowardsNegative (in module bigfloat), 21
RoundTowardsPositive (in module bigfloat), 21
RoundTowardsZero (in module bigfloat), 21

S
sec() (in module bigfloat), 26
sech() (in module bigfloat), 28
set_flag() (in module bigfloat), 33
set_flagstate() (in module bigfloat), 33
setcontext() (in module bigfloat), 22
sgn() (in module bigfloat), 31
sin() (in module bigfloat), 26
single_precision (in module bigfloat), 21
sinh() (in module bigfloat), 28
 sqr() (in module bigfloat), 25
sqrt() (in module bigfloat), 25
sub() (in module bigfloat), 23
subnormalize (bigfloat.Context attribute), 20

T
tan() (in module bigfloat), 26
tanh() (in module bigfloat), 28
test_flag() (in module bigfloat), 33
trunc() (in module bigfloat), 31
U
Underflow (in module bigfloat), 33
unordered() (in module bigfloat), 32

Y
y0() (in module bigfloat), 29
y1() (in module bigfloat), 29
yn() (in module bigfloat), 29

Z
ZeroDivision (in module bigfloat), 33
zeta() (in module bigfloat), 28
zeta_ui() (in module bigfloat), 28