big

package standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 12 | Imported by: 99,448

Details Valid go.mod file ? Redistributable license ? Tagged version ? Stable version ? Learn more

Repository cs.opensource.google/go/go

Links Report a Vulnerability

Documentation

Rendered for linux/amd64

Overview

Package big implements arbitrary-precision arithmetic (big numbers). The following numeric types are supported:

```
Int    signed integers
Rat    rational numbers
Float  floating-point numbers
```

The zero value for an Int, Rat, or Float correspond to 0. Thus, new values can be declared in the usual ways and denote 0 without further initialization:

```
var x Int        // &x is an *Int of value 0
var r = &Rat{}   // r is a *Rat of value 0
y := new(Float)  // y is a *Float of value 0
```

Alternatively, new values can be allocated and initialized with factory functions of the form:

```
func NewT(v V) *T
```

For instance, NewInt(x) returns an *Int set to the value of the int64 argument x, NewRat(a, b) returns a *Rat set to the fraction a/b where a and b are int64 values, and NewFloat(f) returns a *Float initialized to the float64 argument f. More flexibility is provided with explicit setters, for instance:

```
var z1 Int
z1.SetUint64(123)                 // z1 := 123
z2 := new(Rat).SetFloat64(1.25)   // z2 := 5/4
z3 := new(Float).SetInt(z1)       // z3 := 123.0
```

Setters, numeric operations and predicates are represented as methods of the form:

```
func (z *T) SetV(v V) *T          // z = v
func (z *T) Unary(x *T) *T        // z = unary x
func (z *T) Binary(x, y *T) *T    // z = x binary y
func (x *T) Pred() P              // p = pred(x)
```

with T one of Int, Rat, or Float. For unary and binary operations, the result is the receiver (usually named z in that case; see below); if it is one of the operands x or y it may be safely overwritten (and its memory reused).

Arithmetic expressions are typically written as a sequence of individual method calls, with each call corresponding to an operation. The receiver denotes the result and the method arguments are the operation's operands. For instance, given three *Int values a, b and c, the invocation

```
c.Add(a, b)
```

computes the sum a + b and stores the result in c, overwriting whatever value was held in c before. Unless specified otherwise, operations permit aliasing of parameters, so it is perfectly ok to write

```
sum.Add(sum, x)
```

to accumulate values x in a sum.

(By always passing in a result value via the receiver, memory use can be much better controlled. Instead of having to allocate new memory for each result, an operation can reuse the space allocated for the result value, and overwrite that value with the new result in the process.)

Notational convention: Incoming method parameters (including the receiver) are named consistently in the API to clarify their use. Incoming operands are usually named x, y, a, b, and so on, but never z. A parameter specifying the result is named z (typically the receiver).

For instance, the arguments for (*Int).Add are named x and y, and because the receiver specifies the result destination, it is called z:

```
func (z *Int) Add(x, y *Int) *Int
```

Methods of this form typically return the incoming receiver as well, to enable simple call chaining.

Methods which don't require a result value to be passed in (for instance, Int.Sign), simply return the result. In this case, the receiver is typically the first operand, named x:

```
func (x *Int) Sign() int
```

Various methods support conversions between strings and corresponding numeric values, and vice versa: *Int, *Rat, and *Float values implement the Stringer interface for a (default) string representation of the value, but also provide SetString methods to initialize a value from a string in a variety of supported formats (see the respective SetString documentation).

Finally, *Int, *Rat, and *Float satisfy the fmt package's Scanner interface for scanning and (except for *Rat) the Formatter interface for formatted printing.

▶ Example (EConvergents)

▶ Example (Fibonacci)

▶ Example (Sqrt2)

Index

func (x *Float) Mode() RoundingMode
func (z *Float) Mul(x, y *Float) *Float
func (z *Float) Neg(x *Float) *Float
func (z *Float) Parse(s string, base int) (f *Float, b int, err error)
func (x *Float) Prec() uint
func (z *Float) Quo(x, y *Float) *Float
func (x *Float) Rat(z *Rat) (*Rat, Accuracy)
func (z *Float) Scan(s fmt.ScanState, ch rune) error
func (z *Float) Set(x *Float) *Float
func (z *Float) SetFloat64(x float64) *Float
func (z *Float) SetInf(signbit bool) *Float
func (z *Float) SetInt(x *Int) *Float
func (z *Float) SetInt64(x int64) *Float
func (z *Float) SetMantExp(mant *Float, exp int) *Float
func (z *Float) SetMode(mode RoundingMode) *Float
func (z *Float) SetPrec(prec uint) *Float
func (z *Float) SetRat(x *Rat) *Float
func (z *Float) SetString(s string) (*Float, bool)
func (z *Float) SetUint64(x uint64) *Float
func (x *Float) Sign() int
func (x *Float) Signbit() bool
func (z *Float) Sqrt(x *Float) *Float
func (x *Float) String() string
func (z *Float) Sub(x, y *Float) *Float
func (x *Float) Text(format byte, prec int) string
func (x *Float) Uint64() (uint64, Accuracy)
func (z *Float) UnmarshalText(text []byte) error

type Int

func NewInt(x int64) *Int
func (z *Int) Abs(x *Int) *Int
func (z *Int) Add(x, y *Int) *Int
func (z *Int) And(x, y *Int) *Int
func (z *Int) AndNot(x, y *Int) *Int
func (x *Int) Append(buf []byte, base int) []byte
func (z *Int) Binomial(n, k int64) *Int
func (x *Int) Bit(i int) uint
func (x *Int) BitLen() int
func (x *Int) Bits() []Word
func (x *Int) Bytes() []byte
func (x *Int) Cmp(y *Int) (r int)
func (x *Int) CmpAbs(y *Int) int
func (z *Int) Div(x, y *Int) *Int
func (z *Int) DivMod(x, y, m *Int) (*Int, *Int)
func (z *Int) Exp(x, y, m *Int) *Int
func (x *Int) FillBytes(buf []byte) []byte
func (x *Int) Format(s fmt.State, ch rune)

func (z *Int) GCD(x, y, a, b *Int) *Int

func (z *Int) GobDecode(buf []byte) error

func (x *Int) GobEncode() ([]byte, error)

func (x *Int) Int64() int64

func (x *Int) IsInt64() bool

func (x *Int) IsUint64() bool

func (z *Int) Lsh(x *Int, n uint) *Int

func (x *Int) MarshalJSON() ([]byte, error)

func (x *Int) MarshalText() (text []byte, err error)

func (z *Int) Mod(x, y *Int) *Int

func (z *Int) ModInverse(g, n *Int) *Int

func (z *Int) ModSqrt(x, p *Int) *Int

func (z *Int) Mul(x, y *Int) *Int

func (z *Int) MulRange(a, b int64) *Int

func (z *Int) Neg(x *Int) *Int

func (z *Int) Not(x *Int) *Int

func (z *Int) Or(x, y *Int) *Int

func (x *Int) ProbablyPrime(n int) bool

func (z *Int) Quo(x, y *Int) *Int

func (z *Int) QuoRem(x, y, r *Int) (*Int, *Int)

func (z *Int) Rand(rnd *rand.Rand, n *Int) *Int

func (z *Int) Rem(x, y *Int) *Int

func (z *Int) Rsh(x *Int, n uint) *Int

func (z *Int) Scan(s fmt.ScanState, ch rune) error

func (z *Int) Set(x *Int) *Int

func (z *Int) SetBit(x *Int, i int, b uint) *Int

func (z *Int) SetBits(abs []Word) *Int

func (z *Int) SetBytes(buf []byte) *Int

func (z *Int) SetInt64(x int64) *Int

func (z *Int) SetString(s string, base int) (*Int, bool)

func (z *Int) SetUint64(x uint64) *Int

func (x *Int) Sign() int

func (z *Int) Sqrt(x *Int) *Int

func (x *Int) String() string

func (z *Int) Sub(x, y *Int) *Int

func (x *Int) Text(base int) string

func (x *Int) TrailingZeroBits() uint

func (x *Int) Uint64() uint64

func (z *Int) UnmarshalJSON(text []byte) error

func (z *Int) UnmarshalText(text []byte) error

func (z *Int) Xor(x, y *Int) *Int

type Rat

func NewRat(a, b int64) *Rat

func (z *Rat) Abs(x *Rat) *Rat

func (z *Rat) Add(x, y *Rat) *Rat

func (x *Rat) Cmp(y *Rat) int

Examples

Constants

View Source

```
const (
    MaxExp  = math.MaxInt32  // largest supported exponent
    MinExp  = math.MinInt32  // smallest supported exponent
    MaxPrec = math.MaxUint32 // largest (theoretically) supported precision; likely mem
)
```

Exponent and precision limits.

View Source

```
const MaxBase = 10 + ('z' - 'a' + 1) + ('Z' - 'A' + 1)
```

MaxBase is the largest number base accepted for string conversions.

Variables

This section is empty.

Functions

func Jacobi

added in go1.5

```
func Jacobi(x, y *Int) int
```

Jacobi returns the Jacobi symbol (x/y), either +1, -1, or 0. The y argument must be an odd integer.

Types

type Accuracy

added in go1.5

```
type Accuracy int8
```

Accuracy describes the rounding error produced by the most recent operation that generated a Float value, relative to the exact value.

```
const (
    Below Accuracy = -1
    Exact Accuracy = 0
    Above Accuracy = +1
)
```

Constants describing the Accuracy of a Float.

func (Accuracy) String

added in go1.5

```
func (i Accuracy) String() string
```

type ErrNaN

added in go1.5

```
type ErrNaN struct {
	// contains filtered or unexported fields
}
```

An ErrNaN panic is raised by a Float operation that would lead to a NaN under IEEE-754 rules. An ErrNaN implements the error interface.

func (ErrNaN) Error

added in go1.5

```
func (err ErrNaN) Error() string
```

type Float

added in go1.5

```
type Float struct {
	// contains filtered or unexported fields
}
```

A nonzero finite Float represents a multi-precision floating point number

```
sign × mantissa × 2**exponent
```

with 0.5 <= mantissa < 1.0, and MinExp <= exponent <= MaxExp. A Float may also be zero (+0, -0) or infinite (+Inf, -Inf). All Floats are ordered, and the ordering of two Floats x and y is defined by x.Cmp(y).

Each Float value also has a precision, rounding mode, and accuracy. The precision is the maximum number of mantissa bits available to represent the value. The rounding mode specifies how a result should be rounded to fit into the mantissa bits, and accuracy describes the rounding error with respect to the exact result.

Unless specified otherwise, all operations (including setters) that specify a *Float variable for the result (usually via the receiver with the exception of MantExp), round the numeric result according to the precision and rounding mode of the result variable.

If the provided result precision is 0 (see below), it is set to the precision of the argument with the largest precision value before any rounding takes place, and the rounding mode remains unchanged. Thus, uninitialized Floats provided as result arguments will have their precision set to a reasonable value determined by the operands, and their mode is the zero value for RoundingMode (ToNearestEven).

By setting the desired precision to 24 or 53 and using matching rounding mode (typically ToNearestEven), Float operations produce the same results as the corresponding float32 or float64 IEEE-754 arithmetic for operands that correspond to normal (i.e., not denormal) float32 or float64 numbers. Exponent underflow and overflow lead to a 0 or an Infinity for different values than IEEE-754 because Float exponents have a much larger range.

The zero (uninitialized) value for a Float is ready to use and represents the number +0.0 exactly, with precision 0 and rounding mode ToNearestEven.

Operations always take pointer arguments (*Float) rather than Float values, and each unique Float value requires its own unique *Float pointer. To "copy" a Float value, an existing (or newly allocated) Float must be set to a new value using the Float.Set method; shallow copies of Floats are not supported and may lead to errors.

▶ Example (Shift)

func NewFloat

added in go1.5

```
func NewFloat(x float64) *Float
```

NewFloat allocates and returns a new Float set to x, with precision 53 and rounding mode ToNearestEven. NewFloat panics with ErrNaN if x is a NaN.

func ParseFloat

added in go1.5

```
func ParseFloat(s string, base int, prec uint, mode RoundingMode) (f *Float, b int, er
r error)
```

ParseFloat is like f.Parse(s, base) with f set to the given precision and rounding mode.

func (*Float) Abs

added in go1.5

```
func (z *Float) Abs(x *Float) *Float
```

Abs sets z to the (possibly rounded) value |x| (the absolute value of x) and returns z.

func (*Float) Acc

added in go1.5

```
func (x *Float) Acc() Accuracy
```

Acc returns the accuracy of x produced by the most recent operation, unless explicitly documented otherwise by that operation.

func (*Float) Add

added in go1.5

```
func (z *Float) Add(x, y *Float) *Float
```

Add sets z to the rounded sum x+y and returns z. If z's precision is 0, it is changed to the larger of x's or y's precision before the operation. Rounding is performed according to z's precision and rounding mode; and z's accuracy reports the result error relative to the exact (not rounded) result. Add panics with ErrNaN if x and y are infinities with opposite signs. The value of z is undefined in that case.

▶ Example

func (*Float) Append

added in go1.5

```
func (x *Float) Append(buf []byte, fmt byte, prec int) []byte
```

Append appends to buf the string form of the floating-point number x, as generated by x.Text, and returns the extended buffer.

func (*Float) Cmp

added in go1.5

```
func (x *Float) Cmp(y *Float) int
```

Cmp compares x and y and returns:

```
-1 if x <  y
 0 if x == y (incl. -0 == 0, -Inf == -Inf, and +Inf == +Inf)
+1 if x >  y
```

▶ Example

func (*Float) Copy

added in go1.5

```
func (z *Float) Copy(x *Float) *Float
```

Copy sets z to x, with the same precision, rounding mode, and accuracy as x, and returns z. x is not changed even if z and x are the same.

func (*Float) Float32

added in go1.5

```
func (x *Float) Float32() (float32, Accuracy)
```

Float32 returns the float32 value nearest to x. If x is too small to be represented by a float32 (|x| < math.SmallestNonzeroFloat32), the result is (0, Below) or (-0, Above), respectively, depending on the sign of x. If x is too large to be represented by a float32 (|x| > math.MaxFloat32), the result is (+Inf, Above) or (-Inf, Below), depending on the sign of x.

func (*Float) Float64

added in go1.5

```
func (x *Float) Float64() (float64, Accuracy)
```

Float64 returns the float64 value nearest to x. If x is too small to be represented by a float64 (|x| < math.SmallestNonzeroFloat64), the result is (0, Below) or (-0, Above), respectively, depending on the sign of x. If x is too large to be represented by a float64 (|x| > math.MaxFloat64), the result is (+Inf, Above) or (-Inf, Below), depending on the sign of x.

func (*Float) Format

added in go1.5

```
func (x *Float) Format(s fmt.State, format rune)
```

Format implements fmt.Formatter. It accepts all the regular formats for floating-point numbers ('b', 'e', 'E', 'f', 'F', 'g', 'G', 'x') as well as 'p' and 'v'. See (*Float).Text for the interpretation of 'p'. The 'v' format is handled like 'g'. Format also supports specification of the minimum precision in digits, the output field width, as well as the format flags '+' and ' ' for sign control, '0' for space or zero padding, and '-' for left or right justification. See the fmt package for details.

func (*Float) GobDecode

added in go1.7

```
func (z *Float) GobDecode(buf []byte) error
```

GobDecode implements the gob.GobDecoder interface. The result is rounded per the precision and rounding mode of z unless z's precision is 0, in which case z is set exactly to the decoded value.

func (*Float) GobEncode

added in go1.7

```
func (x *Float) GobEncode() ([]byte, error)
```

GobEncode implements the gob.GobEncoder interface. The Float value and all its attributes (precision, rounding mode, accuracy) are marshaled.

func (*Float) Int

added in go1.5

```
func (x *Float) Int(z *Int) (*Int, Accuracy)
```

Int returns the result of truncating x towards zero; or nil if x is an infinity. The result is Exact if x.IsInt(); otherwise it is Below for x > 0, and Above for x < 0. If a non-nil *Int argument z is provided, Int stores the result in z instead of allocating a new Int.

func (*Float) Int64

added in go1.5

```
func (x *Float) Int64() (int64, Accuracy)
```

Int64 returns the integer resulting from truncating x towards zero. If math.MinInt64 <= x <= math.MaxInt64, the result is Exact if x is an integer, and Above (x < 0) or Below (x > 0) otherwise. The result is (math.MinInt64, Above) for x < math.MinInt64, and (math.MaxInt64, Below) for x > math.MaxInt64.

func (*Float) IsInf

added in go1.5

```
func (x *Float) IsInf() bool
```

IsInf reports whether x is +Inf or -Inf.

func (*Float) IsInt

added in go1.5

```
func (x *Float) IsInt() bool
```

IsInt reports whether x is an integer. ±Inf values are not integers.

func (*Float) MantExp

added in go1.5

```
func (x *Float) MantExp(mant *Float) (exp int)
```

MantExp breaks x into its mantissa and exponent components and returns the exponent. If a non-nil mant argument is provided its value is set to the mantissa of x, with the same precision and rounding mode as x. The components satisfy x == mant × 2**exp, with 0.5 <= |mant| < 1.0. Calling MantExp with a nil argument is an efficient way to get the exponent of the receiver.

Special cases are:

```
(  ±0).MantExp(mant) = 0, with mant set to   ±0
(±Inf).MantExp(mant) = 0, with mant set to ±Inf
```

x and mant may be the same in which case x is set to its mantissa value.

func (*Float) MarshalText

added in go1.6

```
func (x *Float) MarshalText() (text []byte, err error)
```

MarshalText implements the encoding.TextMarshaler interface. Only the Float value is marshaled (in full precision), other attributes such as precision or accuracy are ignored.

func (*Float) MinPrec

added in go1.5

```
func (x *Float) MinPrec() uint
```

MinPrec returns the minimum precision required to represent x exactly (i.e., the smallest prec before x.SetPrec(prec) would start rounding x). The result is 0 for |x| == 0 and |x| == Inf.

func (*Float) Mode

added in go1.5

```
func (x *Float) Mode() RoundingMode
```

Mode returns the rounding mode of x.

func (*Float) Mul

added in go1.5

```
func (z *Float) Mul(x, y *Float) *Float
```

Mul sets z to the rounded product x*y and returns z. Precision, rounding, and accuracy reporting are as for Add. Mul panics with ErrNaN if one operand is zero and the other operand an infinity. The value of z is undefined in that case.

func (*Float) Neg

added in go1.5

```
func (z *Float) Neg(x *Float) *Float
```

Neg sets z to the (possibly rounded) value of x with its sign negated, and returns z.

func (*Float) Parse

added in go1.5

```
func (z *Float) Parse(s string, base int) (f *Float, b int, err error)
```

Parse parses s which must contain a text representation of a floating- point number with a mantissa in the given conversion base (the exponent is always a decimal number), or a string representing an infinite value.

For base 0, an underscore character “_” may appear between a base prefix and an adjacent digit, and between successive digits; such underscores do not change the value of the number, or the returned digit count. Incorrect placement of underscores is reported as an error if there are no other errors. If base != 0, underscores are not recognized and thus terminate scanning like any other character that is not a valid radix point or digit.

It sets z to the (possibly rounded) value of the corresponding floating- point value, and returns z, the actual base b, and an error err, if any. The entire string (not just a prefix) must be consumed for success. If z's precision is 0, it is changed to 64 before rounding takes effect. The number must be of the form:

```
number    = [ sign ] ( float | "inf" | "Inf" ) .
sign      = "+" | "-" .
float     = ( mantissa | prefix pmantissa ) [ exponent ] .
prefix    = "0" [ "b" | "B" | "o" | "O" | "x" | "X" ] .
mantissa  = digits "." [ digits ] | digits | "." digits .
pmantissa = [ "_" ] digits "." [ digits ] | [ "_" ] digits | "." digits .
exponent  = ( "e" | "E" | "p" | "P" ) [ sign ] digits .
digits    = digit { [ "_" ] digit } .
digit     = "0" ... "9" | "a" ... "z" | "A" ... "Z" .
```

The base argument must be 0, 2, 8, 10, or 16. Providing an invalid base argument will lead to a run-time panic.

For base 0, the number prefix determines the actual base: A prefix of “0b” or “0B” selects base 2, “0o” or “0O” selects base 8, and “0x” or “0X” selects base 16. Otherwise, the actual base is 10 and no prefix is accepted. The octal prefix "0" is not supported (a leading "0" is simply considered a "0").

A "p" or "P" exponent indicates a base 2 (rather then base 10) exponent; for instance, "0x1.fffffffffffffp1023" (using base 0) represents the maximum float64 value. For hexadecimal mantissae, the exponent character must be one of 'p' or 'P', if present (an "e" or "E" exponent indicator cannot be distinguished from a mantissa digit).

The returned *Float f is nil and the value of z is valid but not defined if an error is reported.

func (*Float) Prec

added in go1.5

```
func (x *Float) Prec() uint
```

Prec returns the mantissa precision of x in bits. The result may be 0 for |x| == 0 and |x| == Inf.

func (*Float) Quo

added in go1.5

```
func (z *Float) Quo(x, y *Float) *Float
```

Quo sets z to the rounded quotient x/y and returns z. Precision, rounding, and accuracy reporting are as for Add. Quo panics with ErrNaN if both operands are zero or infinities. The value of z is undefined in that case.

func (*Float) Rat

added in go1.5

```
func (x *Float) Rat(z *Rat) (*Rat, Accuracy)
```

Rat returns the rational number corresponding to x; or nil if x is an infinity. The result is Exact if x is not an Inf. If a non-nil *Rat argument z is provided, Rat stores the result in z instead of allocating a new Rat.

func (*Float) Scan

added in go1.8

```
func (z *Float) Scan(s fmt.ScanState, ch rune) error
```

Scan is a support routine for fmt.Scanner; it sets z to the value of the scanned number. It accepts formats whose verbs are supported by fmt.Scan for floating point values, which are: 'b' (binary), 'e', 'E', 'f', 'F', 'g' and 'G'. Scan doesn't handle ±Inf.

▶ Example

func (*Float) Set

added in go1.5

```
func (z *Float) Set(x *Float) *Float
```

Set sets z to the (possibly rounded) value of x and returns z. If z's precision is 0, it is changed to the precision of x before setting z (and rounding will have no effect). Rounding is performed according to z's precision and rounding mode; and z's accuracy reports the result error relative to the exact (not rounded) result.

func (*Float) SetFloat64

added in go1.5

```
func (z *Float) SetFloat64(x float64) *Float
```

SetFloat64 sets z to the (possibly rounded) value of x and returns z. If z's precision is 0, it is changed to 53 (and rounding will have no effect). SetFloat64 panics with ErrNaN if x is a NaN.

func (*Float) SetInf

added in go1.5

```
func (z *Float) SetInf(signbit bool) *Float
```

SetInf sets z to the infinite Float -Inf if signbit is set, or +Inf if signbit is not set, and returns z. The precision of z is unchanged and the result is always Exact.

func (*Float) SetInt

added in go1.5

```
func (z *Float) SetInt(x *Int) *Float
```

SetInt sets z to the (possibly rounded) value of x and returns z. If z's precision is 0, it is changed to the larger of x.BitLen() or 64 (and rounding will have no effect).

func (*Float) SetInt64

added in go1.5

```
func (z *Float) SetInt64(x int64) *Float
```

SetInt64 sets z to the (possibly rounded) value of x and returns z. If z's precision is 0, it is changed to 64 (and rounding will have no effect).

func (*Float) SetMantExp

added in go1.5

```
func (z *Float) SetMantExp(mant *Float, exp int) *Float
```

SetMantExp sets z to mant × 2**exp and returns z. The result z has the same precision and rounding mode as mant. SetMantExp is an inverse of MantExp but does not require 0.5 <= |mant| < 1.0. Specifically, for a given x of type *Float, SetMantExp relates to MantExp as follows:

```
mant := new(Float)
new(Float).SetMantExp(mant, x.MantExp(mant)).Cmp(x) == 0
```

Special cases are:

```
z.SetMantExp(  ±0, exp) =   ±0
z.SetMantExp(±Inf, exp) = ±Inf
```

z and mant may be the same in which case z's exponent is set to exp.

func (*Float) SetMode

added in go1.5

```
func (z *Float) SetMode(mode RoundingMode) *Float
```

SetMode sets z's rounding mode to mode and returns an exact z. z remains unchanged otherwise. z.SetMode(z.Mode()) is a cheap way to set z's accuracy to Exact.

func (*Float) SetPrec

added in go1.5

```
func (z *Float) SetPrec(prec uint) *Float
```

SetPrec sets z's precision to prec and returns the (possibly) rounded value of z. Rounding occurs according to z's rounding mode if the mantissa cannot be represented in prec bits without loss of precision. SetPrec(0) maps all finite values to ±0; infinite values remain unchanged. If prec > MaxPrec, it is set to MaxPrec.

func (*Float) SetRat

added in go1.5

```
func (z *Float) SetRat(x *Rat) *Float
```

SetRat sets z to the (possibly rounded) value of x and returns z. If z's precision is 0, it is changed to the largest of a.BitLen(), b.BitLen(), or 64; with x = a/b.

func (*Float) SetString

added in go1.5

```
func (z *Float) SetString(s string) (*Float, bool)
```

SetString sets z to the value of s and returns z and a boolean indicating success. s must be a floating-point number of the same format as accepted by Parse, with base argument 0. The entire string (not just a prefix) must be valid for success. If the operation failed, the value of z is undefined but the returned value is nil.

▶ Example

func (*Float) SetUint64

added in go1.5

```
func (z *Float) SetUint64(x uint64) *Float
```

SetUint64 sets z to the (possibly rounded) value of x and returns z. If z's precision is 0, it is changed to 64 (and rounding will have no effect).

func (*Float) Sign

added in go1.5

```
func (x *Float) Sign() int
```

Sign returns:

```
-1 if x <   0
 0 if x is ±0
+1 if x >   0
```

func (*Float) Signbit

added in go1.5

```
func (x *Float) Signbit() bool
```

Signbit reports whether x is negative or negative zero.

func (*Float) Sqrt

added in go1.10

```
func (z *Float) Sqrt(x *Float) *Float
```

Sqrt sets z to the rounded square root of x, and returns it.

If z's precision is 0, it is changed to x's precision before the operation. Rounding is performed according to z's precision and rounding mode, but z's accuracy is not computed. Specifically, the result of z.Acc() is undefined.

The function panics if z < 0. The value of z is undefined in that case.

func (*Float) String

added in go1.5

```
func (x *Float) String() string
```

String formats x like x.Text('g', 10). (String must be called explicitly, Float.Format does not support %s verb.)

func (*Float) Sub

added in go1.5

```
func (z *Float) Sub(x, y *Float) *Float
```

Sub sets z to the rounded difference x-y and returns z. Precision, rounding, and accuracy reporting are as for Add. Sub panics with ErrNaN if x and y are infinities with equal signs. The value of z is undefined in that case.

func (*Float) Text

added in go1.5

```
func (x *Float) Text(format byte, prec int) string
```

Text converts the floating-point number x to a string according to the given format and precision prec. The format is one of:

```
'e' -d.dddde±dd, decimal exponent, at least two (possibly 0) exponent digits
'E' -d.ddddE±dd, decimal exponent, at least two (possibly 0) exponent digits
'f' -ddddd.dddd, no exponent
'g' like 'e' for large exponents, like 'f' otherwise
'G' like 'E' for large exponents, like 'f' otherwise
'x' -0xd.dddddp±dd, hexadecimal mantissa, decimal power of two exponent
'p' -0x.dddp±dd, hexadecimal mantissa, decimal power of two exponent (non-standard)
'b' -ddddddp±dd, decimal mantissa, decimal power of two exponent (non-standard)
```

For the power-of-two exponent formats, the mantissa is printed in normalized form:

```
'x' hexadecimal mantissa in [1, 2), or 0
'p' hexadecimal mantissa in [½, 1), or 0
'b' decimal integer mantissa using x.Prec() bits, or 0
```

Note that the 'x' form is the one used by most other languages and libraries.

If format is a different character, Text returns a "%" followed by the unrecognized format character.

The precision prec controls the number of digits (excluding the exponent) printed by the 'e', 'E', 'f', 'g', 'G', and 'x' formats. For 'e', 'E', 'f', and 'x', it is the number of digits after the decimal point. For 'g' and 'G' it is the total number of digits. A negative precision selects the smallest number of decimal digits necessary to identify the value x uniquely using x.Prec() mantissa bits. The prec value is ignored for the 'b' and 'p' formats.

func (*Float) Uint64

added in go1.5

```
func (x *Float) Uint64() (uint64, Accuracy)
```

Uint64 returns the unsigned integer resulting from truncating x towards zero. If 0 <= x <= math.MaxUint64, the result is Exact if x is an integer and Below otherwise. The result is (0, Above) for x < 0, and (math.MaxUint64, Below) for x > math.MaxUint64.

func (*Float) UnmarshalText

added in go1.6

```
func (z *Float) UnmarshalText(text []byte) error
```

UnmarshalText implements the encoding.TextUnmarshaler interface. The result is rounded per the precision and rounding mode of z. If z's precision is 0, it is changed to 64 before rounding takes effect.

type Int

```
type Int struct {
	// contains filtered or unexported fields
}
```

An Int represents a signed multi-precision integer. The zero value for an Int represents the value 0.

Operations always take pointer arguments (*Int) rather than Int values, and each unique Int value requires its own unique *Int pointer. To "copy" an Int value, an existing (or newly allocated) Int must be set to a new value using the Int.Set method; shallow copies of Ints are not supported and may lead to errors.

func NewInt

```
func NewInt(x int64) *Int
```

NewInt allocates and returns a new Int set to x.

func (*Int) Abs

```
func (z *Int) Abs(x *Int) *Int
```

Abs sets z to |x| (the absolute value of x) and returns z.

func (*Int) Add

```
func (z *Int) Add(x, y *Int) *Int
```

Add sets z to the sum x+y and returns z.

func (*Int) And

```
func (z *Int) And(x, y *Int) *Int
```

And sets z = x & y and returns z.

func (*Int) AndNot

```
func (z *Int) AndNot(x, y *Int) *Int
```

AndNot sets z = x &^ y and returns z.

func (*Int) Append

added in go1.6

```
func (x *Int) Append(buf []byte, base int) []byte
```

Append appends the string representation of x, as generated by x.Text(base), to buf and returns the extended buffer.

func (*Int) Binomial

```
func (z *Int) Binomial(n, k int64) *Int
```

Binomial sets z to the binomial coefficient C(n, k) and returns z.

func (*Int) Bit

```
func (x *Int) Bit(i int) uint
```

Bit returns the value of the i'th bit of x. That is, it returns (x>>i)&1. The bit index i must be >= 0.

func (*Int) BitLen

```
func (x *Int) BitLen() int
```

BitLen returns the length of the absolute value of x in bits. The bit length of 0 is 0.

func (*Int) Bits

```
func (x *Int) Bits() []Word
```

Bits provides raw (unchecked but fast) access to x by returning its absolute value as a little-endian Word slice. The result and x share the same underlying array. Bits is intended to support implementation of missing low-level Int functionality outside this package; it should be avoided otherwise.

func (*Int) Bytes

```
func (x *Int) Bytes() []byte
```

Bytes returns the absolute value of x as a big-endian byte slice.

To use a fixed length slice, or a preallocated one, use FillBytes.

func (*Int) Cmp

```
func (x *Int) Cmp(y *Int) (r int)
```

Cmp compares x and y and returns:

```
-1 if x <  y
 0 if x == y
+1 if x >  y
```

func (*Int) CmpAbs

added in go1.10

```
func (x *Int) CmpAbs(y *Int) int
```

CmpAbs compares the absolute values of x and y and returns:

```
-1 if |x| <  |y|
 0 if |x| == |y|
+1 if |x| >  |y|
```

func (*Int) Div

```
func (z *Int) Div(x, y *Int) *Int
```

Div sets z to the quotient x/y for y != 0 and returns z. If y == 0, a division-by-zero run-time panic occurs. Div implements Euclidean division (unlike Go); see DivMod for more details.

func (*Int) DivMod

```
func (z *Int) DivMod(x, y, m *Int) (*Int, *Int)
```

DivMod sets z to the quotient x div y and m to the modulus x mod y and returns the pair (z, m) for y != 0. If y == 0, a division-by-zero run-time panic occurs.

DivMod implements Euclidean division and modulus (unlike Go):

```
q = x div y  such that
m = x - y*q  with 0 <= m < |y|
```

(See Raymond T. Boute, “The Euclidean definition of the functions div and mod”. ACM Transactions on Programming Languages and Systems (TOPLAS), 14(2):127-144, New York, NY, USA, 4/1992. ACM press.) See QuoRem for T-division and modulus (like Go).

func (*Int) Exp

```
func (z *Int) Exp(x, y, m *Int) *Int
```

Exp sets z = x**y mod |m| (i.e. the sign of m is ignored), and returns z. If m == nil or m == 0, z = x**y unless y <= 0 then z = 1. If m != 0, y < 0, and x and m are not relatively prime, z is unchanged and nil is returned.

Modular exponentiation of inputs of a particular size is not a cryptographically constant-time operation.

func (*Int) FillBytes

added in go1.15

```
func (x *Int) FillBytes(buf []byte) []byte
```

FillBytes sets buf to the absolute value of x, storing it as a zero-extended big-endian byte slice, and returns buf.

If the absolute value of x doesn't fit in buf, FillBytes will panic.

func (*Int) Format

```
func (x *Int) Format(s fmt.State, ch rune)
```

Format implements fmt.Formatter. It accepts the formats 'b' (binary), 'o' (octal with 0 prefix), 'O' (octal with 0o prefix), 'd' (decimal), 'x' (lowercase hexadecimal), and 'X' (uppercase hexadecimal). Also supported are the full suite of package fmt's format flags for integral types, including '+' and ' ' for sign control, '#' for leading zero in octal and for hexadecimal, a leading "0x" or "0X" for "%#x" and "%#X" respectively, specification of minimum digits precision, output field width, space or zero padding, and '-' for left or right justification.

func (*Int) GCD

```
func (z *Int) GCD(x, y, a, b *Int) *Int
```

GCD sets z to the greatest common divisor of a and b and returns z. If x or y are not nil, GCD sets their value such that z = a*x + b*y.

a and b may be positive, zero or negative. (Before Go 1.14 both had to be > 0.) Regardless of the signs of a and b, z is always >= 0.

If a == b == 0, GCD sets z = x = y = 0.

If a == 0 and b != 0, GCD sets z = |b|, x = 0, y = sign(b) * 1.

If a != 0 and b == 0, GCD sets z = |a|, x = sign(a) * 1, y = 0.

func (*Int) GobDecode

```
func (z *Int) GobDecode(buf []byte) error
```

GobDecode implements the gob.GobDecoder interface.

func (*Int) GobEncode

```
func (x *Int) GobEncode() ([]byte, error)
```

GobEncode implements the gob.GobEncoder interface.

func (*Int) Int64

```
func (x *Int) Int64() int64
```

Int64 returns the int64 representation of x. If x cannot be represented in an int64, the result is undefined.

func (*Int) IsInt64

added in go1.9

```
func (x *Int) IsInt64() bool
```

IsInt64 reports whether x can be represented as an int64.

func (*Int) IsUint64

added in go1.9

```
func (x *Int) IsUint64() bool
```

IsUint64 reports whether x can be represented as a uint64.

func (*Int) Lsh

```
func (z *Int) Lsh(x *Int, n uint) *Int
```

Lsh sets z = x << n and returns z.

func (*Int) MarshalJSON

added in go1.1

```
func (x *Int) MarshalJSON() ([]byte, error)
```

MarshalJSON implements the json.Marshaler interface.

func (*Int) MarshalText

added in go1.3

```
func (x *Int) MarshalText() (text []byte, err error)
```

MarshalText implements the encoding.TextMarshaler interface.

func (*Int) Mod

```
func (z *Int) Mod(x, y *Int) *Int
```

Mod sets z to the modulus x%y for y != 0 and returns z. If y == 0, a division-by-zero run-time panic occurs. Mod implements Euclidean modulus (unlike Go); see DivMod for more details.

func (*Int) ModInverse

```
func (z *Int) ModInverse(g, n *Int) *Int
```

ModInverse sets z to the multiplicative inverse of g in the ring ℤ/nℤ and returns z. If g and n are not relatively prime, g has no multiplicative inverse in the ring ℤ/nℤ. In this case, z is unchanged and the return value is nil. If n == 0, a division-by-zero run-time panic occurs.

func (*Int) ModSqrt

added in go1.5

```
func (z *Int) ModSqrt(x, p *Int) *Int
```

ModSqrt sets z to a square root of x mod p if such a square root exists, and returns z. The modulus p must be an odd prime. If x is not a square mod p, ModSqrt leaves z unchanged and returns nil. This function panics if p is not an odd integer, its behavior is undefined if p is odd but not prime.

func (*Int) Mul

```
func (z *Int) Mul(x, y *Int) *Int
```

Mul sets z to the product x*y and returns z.

func (*Int) MulRange

```
func (z *Int) MulRange(a, b int64) *Int
```

MulRange sets z to the product of all integers in the range [a, b] inclusively and returns z. If a > b (empty range), the result is 1.

func (*Int) Neg

```
func (z *Int) Neg(x *Int) *Int
```

Neg sets z to -x and returns z.

func (*Int) Not

```
func (z *Int) Not(x *Int) *Int
```

Not sets z = ^x and returns z.

func (*Int) Or

```
func (z *Int) Or(x, y *Int) *Int
```

Or sets z = x | y and returns z.

func (*Int) ProbablyPrime

```
func (x *Int) ProbablyPrime(n int) bool
```

ProbablyPrime reports whether x is probably prime, applying the Miller-Rabin test with n pseudorandomly chosen bases as well as a Baillie-PSW test.

If x is prime, ProbablyPrime returns true. If x is chosen randomly and not prime, ProbablyPrime probably returns false. The probability of returning true for a randomly chosen non-prime is at most $¼^n$.

ProbablyPrime is 100% accurate for inputs less than 2^{64}. See Menezes et al., Handbook of Applied Cryptography, 1997, pp. 145-149, and FIPS 186-4 Appendix F for further discussion of the error probabilities.

ProbablyPrime is not suitable for judging primes that an adversary may have crafted to fool the test.

As of Go 1.8, ProbablyPrime(0) is allowed and applies only a Baillie-PSW test. Before Go 1.8, ProbablyPrime applied only the Miller-Rabin tests, and ProbablyPrime(0) panicked.

func (*Int) Quo

```
func (z *Int) Quo(x, y *Int) *Int
```

Quo sets z to the quotient x/y for y != 0 and returns z. If y == 0, a division-by-zero run-time panic occurs. Quo implements truncated division (like Go); see QuoRem for more details.

func (*Int) QuoRem

```
func (z *Int) QuoRem(x, y, r *Int) (*Int, *Int)
```

QuoRem sets z to the quotient x/y and r to the remainder x%y and returns the pair (z, r) for y != 0. If y == 0, a division-by-zero run-time panic occurs.

QuoRem implements T-division and modulus (like Go):

```
q = x/y      with the result truncated to zero
r = x - y*q
```

(See Daan Leijen, “Division and Modulus for Computer Scientists”.) See DivMod for Euclidean division and modulus (unlike Go).

func (*Int) Rand

```
func (z *Int) Rand(rnd *rand.Rand, n *Int) *Int
```

Rand sets z to a pseudo-random number in [0, n) and returns z.

As this uses the math/rand package, it must not be used for security-sensitive work. Use crypto/rand.Int instead.

func (*Int) Rem

```
func (z *Int) Rem(x, y *Int) *Int
```

Rem sets z to the remainder x%y for y != 0 and returns z. If y == 0, a division-by-zero run-time panic occurs. Rem implements truncated modulus (like Go); see QuoRem for more details.

func (*Int) Rsh

```
func (z *Int) Rsh(x *Int, n uint) *Int
```

Rsh sets z = x >> n and returns z.

func (*Int) Scan

```
func (z *Int) Scan(s fmt.ScanState, ch rune) error
```

Scan is a support routine for fmt.Scanner; it sets z to the value of the scanned number. It accepts the formats 'b' (binary), 'o' (octal), 'd' (decimal), 'x' (lowercase hexadecimal), and 'X' (uppercase hexadecimal).

▶ Example

func (*Int) Set

```
func (z *Int) Set(x *Int) *Int
```

Set sets z to x and returns z.

func (*Int) SetBit

```
func (z *Int) SetBit(x *Int, i int, b uint) *Int
```

SetBit sets z to x, with x's i'th bit set to b (0 or 1). That is, if b is 1 SetBit sets z = x | (1 << i); if b is 0 SetBit sets z = x &^ (1 << i). If b is not 0 or 1, SetBit will panic.

func (*Int) SetBits

```
func (z *Int) SetBits(abs []Word) *Int
```

SetBits provides raw (unchecked but fast) access to z by setting its value to abs, interpreted as a little-endian Word slice, and returning z. The result and abs share the same underlying array. SetBits is intended to support implementation of missing low-level Int functionality outside this package; it should be avoided otherwise.

func (*Int) SetBytes

```
func (z *Int) SetBytes(buf []byte) *Int
```

SetBytes interprets buf as the bytes of a big-endian unsigned integer, sets z to that value, and returns z.

func (*Int) SetInt64

```
func (z *Int) SetInt64(x int64) *Int
```

SetInt64 sets z to x and returns z.

func (*Int) SetString

```
func (z *Int) SetString(s string, base int) (*Int, bool)
```

SetString sets z to the value of s, interpreted in the given base, and returns z and a boolean indicating success. The entire string (not just a prefix) must be valid for success. If SetString fails, the value of z is undefined but the returned value is nil.

The base argument must be 0 or a value between 2 and MaxBase. For base 0, the number prefix determines the actual base: A prefix of “0b” or “0B” selects base 2, “0”, “0o” or “0O” selects base 8, and “0x” or “0X” selects base 16. Otherwise, the selected base is 10 and no prefix is accepted.

For bases <= 36, lower and upper case letters are considered the same: The letters 'a' to 'z' and 'A' to 'Z' represent digit values 10 to 35. For bases > 36, the upper case letters 'A' to 'Z' represent the digit values 36 to 61.

For base 0, an underscore character “_” may appear between a base prefix and an adjacent digit, and between successive digits; such underscores do not change the value of the number. Incorrect placement of underscores is reported as an error if there are no other errors. If base != 0, underscores are not recognized and act like any other character that is not a valid digit.

▶ Example

func (*Int) SetUint64

added in go1.1

```
func (z *Int) SetUint64(x uint64) *Int
```

SetUint64 sets z to x and returns z.

func (*Int) Sign

```
func (x *Int) Sign() int
```

Sign returns:

```
-1 if x <  0
 0 if x == 0
+1 if x >  0
```

func (*Int) Sqrt

added in go1.8

```
func (z *Int) Sqrt(x *Int) *Int
```

Sqrt sets z to ⌊√x⌋, the largest integer such that $z^2 \leq x$, and returns z. It panics if x is negative.

func (*Int) String

```
func (x *Int) String() string
```

String returns the decimal representation of x as generated by x.Text(10).

func (*Int) Sub

```
func (z *Int) Sub(x, y *Int) *Int
```

Sub sets z to the difference x-y and returns z.

func (*Int) Text

added in go1.6

```
func (x *Int) Text(base int) string
```

Text returns the string representation of x in the given base. Base must be between 2 and 62, inclusive. The result uses the lower-case letters 'a' to 'z' for digit values 10 to 35, and the upper-case letters 'A' to 'Z' for digit values 36 to 61. No prefix (such as "0x") is added to the string. If x is a nil pointer it returns "<nil>".

func (*Int) TrailingZeroBits

added in go1.13

```
func (x *Int) TrailingZeroBits() uint
```

TrailingZeroBits returns the number of consecutive least significant zero bits of |x|.

func (*Int) Uint64

added in go1.1

```
func (x *Int) Uint64() uint64
```

Uint64 returns the uint64 representation of x. If x cannot be represented in a uint64, the result is undefined.

func (*Int) UnmarshalJSON

added in go1.1

```
func (z *Int) UnmarshalJSON(text []byte) error
```

UnmarshalJSON implements the json.Unmarshaler interface.

func (*Int) UnmarshalText

added in go1.3

```
func (z *Int) UnmarshalText(text []byte) error
```

UnmarshalText implements the encoding.TextUnmarshaler interface.

func (*Int) Xor

```
func (z *Int) Xor(x, y *Int) *Int
```

Xor sets z = x ^ y and returns z.

type Rat

```
type Rat struct {
	// contains filtered or unexported fields
}
```

A Rat represents a quotient a/b of arbitrary precision. The zero value for a Rat represents the value 0.

Operations always take pointer arguments (*Rat) rather than Rat values, and each unique Rat value requires its own unique *Rat pointer. To "copy" a Rat value, an existing (or newly allocated) Rat must be set to a new value using the Rat.Set method; shallow copies of Rats are not supported and may lead to errors.

func NewRat

```
func NewRat(a, b int64) *Rat
```

NewRat creates a new Rat with numerator a and denominator b.

func (*Rat) Abs

```
func (z *Rat) Abs(x *Rat) *Rat
```

Abs sets z to |x| (the absolute value of x) and returns z.

func (*Rat) Add

```
func (z *Rat) Add(x, y *Rat) *Rat
```

Add sets z to the sum x+y and returns z.

func (*Rat) Cmp

```
func (x *Rat) Cmp(y *Rat) int
```

Cmp compares x and y and returns:

```
-1 if x <  y
 0 if x == y
+1 if x >  y
```

func (*Rat) Denom

```
func (x *Rat) Denom() *Int
```

Denom returns the denominator of x; it is always > 0. The result is a reference to x's denominator, unless x is an uninitialized (zero value) Rat, in which case the result is a new Int of value 1. (To initialize x, any operation that sets x will do, including x.Set(x).) If the result is a reference to x's denominator it may change if a new value is assigned to x, and vice versa.

func (*Rat) Float32

added in go1.4

```
func (x *Rat) Float32() (f float32, exact bool)
```

Float32 returns the nearest float32 value for x and a bool indicating whether f represents x exactly. If the magnitude of x is too large to be represented by a float32, f is an infinity and exact is false. The sign of f always matches the sign of x, even if f == 0.

func (*Rat) Float64

added in go1.1

```
func (x *Rat) Float64() (f float64, exact bool)
```

Float64 returns the nearest float64 value for x and a bool indicating whether f represents x exactly. If the magnitude of x is too large to be represented by a float64, f is an infinity and exact is false. The sign of f always matches the sign of x, even if f == 0.

func (*Rat) FloatString

```
func (x *Rat) FloatString(prec int) string
```

FloatString returns a string representation of x in decimal form with prec digits of precision after the radix point. The last digit is rounded to nearest, with halves rounded away from zero.

func (*Rat) GobDecode

```
func (z *Rat) GobDecode(buf []byte) error
```

GobDecode implements the gob.GobDecoder interface.

func (*Rat) GobEncode

```
func (x *Rat) GobEncode() ([]byte, error)
```

GobEncode implements the gob.GobEncoder interface.

func (*Rat) Inv

```
func (z *Rat) Inv(x *Rat) *Rat
```

Inv sets z to 1/x and returns z. If x == 0, Inv panics.

func (*Rat) IsInt

```
func (x *Rat) IsInt() bool
```

IsInt reports whether the denominator of x is 1.

func (*Rat) MarshalText

added in go1.3

```
func (x *Rat) MarshalText() (text []byte, err error)
```

MarshalText implements the encoding.TextMarshaler interface.

func (*Rat) Mul

```
func (z *Rat) Mul(x, y *Rat) *Rat
```

Mul sets z to the product x*y and returns z.

func (*Rat) Neg

```
func (z *Rat) Neg(x *Rat) *Rat
```

Neg sets z to -x and returns z.

func (*Rat) Num

```
func (x *Rat) Num() *Int
```

Num returns the numerator of x; it may be <= 0. The result is a reference to x's numerator; it may change if a new value is assigned to x, and vice versa. The sign of the numerator corresponds to the sign of x.

func (*Rat) Quo

```
func (z *Rat) Quo(x, y *Rat) *Rat
```

Quo sets z to the quotient x/y and returns z. If y == 0, Quo panics.

func (*Rat) RatString

```
func (x *Rat) RatString() string
```

RatString returns a string representation of x in the form "a/b" if b != 1, and in the form "a" if b == 1.

func (*Rat) Scan

```
func (z *Rat) Scan(s fmt.ScanState, ch rune) error
```

Scan is a support routine for fmt.Scanner. It accepts the formats 'e', 'E', 'f', 'F', 'g', 'G', and 'v'. All formats are equivalent.

▶ Example

func (*Rat) Set

```
func (z *Rat) Set(x *Rat) *Rat
```

Set sets z to x (by making a copy of x) and returns z.

func (*Rat) SetFloat64

added in go1.1

```
func (z *Rat) SetFloat64(f float64) *Rat
```

SetFloat64 sets z to exactly f and returns z. If f is not finite, SetFloat returns nil.

func (*Rat) SetFrac

```
func (z *Rat) SetFrac(a, b *Int) *Rat
```

SetFrac sets z to a/b and returns z. If b == 0, SetFrac panics.

func (*Rat) SetFrac64

```
func (z *Rat) SetFrac64(a, b int64) *Rat
```

SetFrac64 sets z to a/b and returns z. If b == 0, SetFrac64 panics.

func (*Rat) SetInt

```
func (z *Rat) SetInt(x *Int) *Rat
```

SetInt sets z to x (by making a copy of x) and returns z.

func (*Rat) SetInt64

```
func (z *Rat) SetInt64(x int64) *Rat
```

SetInt64 sets z to x and returns z.

func (*Rat) SetString

```
func (z *Rat) SetString(s string) (*Rat, bool)
```

SetString sets z to the value of s and returns z and a boolean indicating success. s can be given as a (possibly signed) fraction "a/b", or as a floating-point number optionally followed by an exponent. If a fraction is provided, both the dividend and the divisor may be a decimal integer or independently use a prefix of “0b”, “0” or “0o”, or “0x” (or their upper-case variants) to denote a binary, octal, or hexadecimal integer, respectively. The divisor may not be signed. If a floating-point number is provided, it may be in decimal form or use any of the same prefixes as above but for “0” to denote a non-decimal mantissa. A leading “0” is considered a decimal leading 0; it does not indicate octal representation in this case. An optional base-10 “e” or base-2 “p” (or their upper-case variants) exponent may be provided as well, except for hexadecimal floats which only accept an (optional) “p” exponent (because an “e” or “E” cannot be distinguished from a mantissa digit). If the exponent's absolute value is too large, the operation may fail. The entire string, not just a prefix, must be valid for success. If the operation failed, the value of z is undefined but the returned value is nil.

▶ Example

func (*Rat) SetUint64

added in go1.13

```
func (z *Rat) SetUint64(x uint64) *Rat
```

SetUint64 sets z to x and returns z.

func (*Rat) Sign

```
func (x *Rat) Sign() int
```

Sign returns:

```
-1 if x <  0
 0 if x == 0
+1 if x >  0
```

func (*Rat) String

```
func (x *Rat) String() string
```

String returns a string representation of x in the form "a/b" (even if b == 1).

func (*Rat) Sub

```
func (z *Rat) Sub(x, y *Rat) *Rat
```

Sub sets z to the difference x-y and returns z.

func (*Rat) UnmarshalText

added in go1.3

```
func (z *Rat) UnmarshalText(text []byte) error
```

UnmarshalText implements the encoding.TextUnmarshaler interface.

type RoundingMode

added in go1.5

```
type RoundingMode byte
```

RoundingMode determines how a Float value is rounded to the desired precision. Rounding may change the Float value; the rounding error is described by the Float's Accuracy.

▶ Example

```
const (
    ToNearestEven RoundingMode = iota // == IEEE 754-2008 roundTiesToEven
    ToNearestAway                     // == IEEE 754-2008 roundTiesToAway
    ToZero                            // == IEEE 754-2008 roundTowardZero
    AwayFromZero                      // no IEEE 754-2008 equivalent
    ToNegativeInf                     // == IEEE 754-2008 roundTowardNegative
    ToPositiveInf                     // == IEEE 754-2008 roundTowardPositive
)
```

These constants define supported rounding modes.

func (RoundingMode) String

added in go1.5

```
func (i RoundingMode) String() string
```

type Word

```
type Word uint
```

A Word represents a single digit of a multi-precision unsigned integer.

Source Files

View all

accuracy_string.go	floatmarsh.go	prime.go
arith.go	ftoa.go	rat.go
arith_amd64.go	int.go	ratconv.go
arith_decl.go	intconv.go	ratmarsh.go
decimal.go	intmarsh.go	roundingmode_string.go
doc.go	nat.go	sqrt.go
float.go	natconv.go	
floatconv.go	natdiv.go	