

[Discover Packages](#) > [Standard library](#) > [strconv](#) 





strconv

package

standard library

Version: [go1.20.1](#) **Latest** | Published: Feb 14, 2023 | License: [BSD-3-Clause](#) | Imports: [5](#) |Imported by: [663,745](#)

Details



- ✓ Valid [go.mod](#) file 
- ✓ Redistributable license 
- ✓ Tagged version 
- ✓ Stable version 

[Learn more](#)

Repository

cs.opensource.google/go/go

Links

 [Report a Vulnerability](#) Documentation 

<> Documentation

Overview

[Numeric Conversions](#)[String Conversions](#)

Package strconv implements conversions to and from string representations of basic data types.

Numeric Conversions

The most common numeric conversions are `Atoi` (string to int) and `Itoa` (int to string).

```
i, err := strconv.Atoi("-42")
s := strconv.Itoa(-42)
```

These assume decimal and the Go int type.

`ParseBool`, `ParseFloat`, `ParseInt`, and `ParseUint` convert strings to values:

```
b, err := strconv.ParseBool("true")
f, err := strconv.ParseFloat("3.1415", 64)
i, err := strconv.ParseInt("-42", 10, 64)
u, err := strconv.ParseUint("42", 10, 64)
```

The parse functions return the widest type (`float64`, `int64`, and `uint64`), but if the size argument specifies a narrower width the result can be converted to that narrower type without data loss:

```
s := "2147483647" // biggest int32
i64, err := strconv.ParseInt(s, 10, 32)
...
i := int32(i64)
```

`FormatBool`, `FormatFloat`, `FormatInt`, and `FormatUint` convert values to strings:

```
s := strconv.FormatBool(true)
s := strconv.FormatFloat(3.1415, 'E', -1, 64)
s := strconv.FormatInt(-42, 16)
s := strconv.FormatUint(42, 16)
```

`AppendBool`, `AppendFloat`, `AppendInt`, and `AppendUint` are similar but append the formatted value to a destination slice.

String Conversions

`Quote` and `QuoteToASCII` convert strings to quoted Go string literals. The latter guarantees that the result is an ASCII string, by escaping any non-ASCII Unicode with `\u`:

```
q := strconv.Quote("Hello, 世界")
q := strconv.QuoteToASCII("Hello, 世界")
```

`QuoteRune` and `QuoteRuneToASCII` are similar but accept runes and return quoted Go rune literals.

`Unquote` and `UnquoteChar` unquote Go string and rune literals.

Index

[Constants](#)

[Variables](#)

[func AppendBool\(dst \[\]byte, b bool\) \[\]byte](#)
[func AppendFloat\(dst \[\]byte, f float64, fmt byte, prec, bitSize int\) \[\]byte](#)
[func AppendInt\(dst \[\]byte, i int64, base int\) \[\]byte](#)
[func AppendQuote\(dst \[\]byte, s string\) \[\]byte](#)
[func AppendQuoteRune\(dst \[\]byte, r rune\) \[\]byte](#)
[func AppendQuoteRuneToASCII\(dst \[\]byte, r rune\) \[\]byte](#)
[func AppendQuoteRuneToGraphic\(dst \[\]byte, r rune\) \[\]byte](#)
[func AppendQuoteToASCII\(dst \[\]byte, s string\) \[\]byte](#)
[func AppendQuoteToGraphic\(dst \[\]byte, s string\) \[\]byte](#)
[func AppendUint\(dst \[\]byte, i uint64, base int\) \[\]byte](#)
[func Atoi\(s string\) \(int, error\)](#)
[func CanBackquote\(s string\) bool](#)
[func FormatBool\(b bool\) string](#)
[func FormatComplex\(c complex128, fmt byte, prec, bitSize int\) string](#)
[func FormatFloat\(f float64, fmt byte, prec, bitSize int\) string](#)
[func FormatInt\(i int64, base int\) string](#)
[func FormatUint\(i uint64, base int\) string](#)

```
func IsGraphic(r rune) bool
func IsPrint(r rune) bool
func Itoa(i int) string
func ParseBool(str string) (bool, error)
func ParseComplex(s string, bitSize int) (complex128, error)
func ParseFloat(s string, bitSize int) (float64, error)
func ParseInt(s string, base int, bitSize int) (i int64, err error)
func ParseUint(s string, base int, bitSize int) (uint64, error)
func Quote(s string) string
func QuoteRune(r rune) string
func QuoteRuneToASCII(r rune) string
func QuoteRuneToGraphic(r rune) string
func QuoteToASCII(s string) string
func QuoteToGraphic(s string) string
func QuotedPrefix(s string) (string, error)
func Unquote(s string) (string, error)
func UnquoteChar(s string, quote byte) (value rune, multibyte bool, tail string, err error)
type NumError
    func (e *NumError) Error() string
    func (e *NumError) Unwrap() error
```

Examples

```
AppendBool
AppendFloat
AppendInt
AppendQuote
AppendQuoteRune
AppendQuoteRuneToASCII
AppendQuoteToASCII
AppendUint
Atoi
CanBackquote
FormatBool
FormatFloat
FormatInt
FormatUint
IsGraphic
IsPrint
Itoa
NumError
ParseBool
ParseFloat
ParseInt
ParseUint
Quote
QuoteRune
```

[QuoteRuneToASCII](#)
[QuoteRuneToGraphic](#)
[QuoteToASCII](#)
[QuoteToGraphic](#)
[Unquote](#)
[UnquoteChar](#)

Constants

[View Source](#)

```
const IntSize = intSize
```

IntSize is the size in bits of an int or uint value.

Variables

[View Source](#)

```
var ErrRange = errors.New("value out of range")
```

ErrRange indicates that a value is out of range for the target type.

[View Source](#)

```
var ErrSyntax = errors.New("invalid syntax")
```

ErrSyntax indicates that a value does not have the right syntax for the target type.

Functions

func [AppendBool](#)

```
func AppendBool(dst []byte, b bool) []byte
```

AppendBool appends "true" or "false", according to the value of b, to dst and returns the extended buffer.

► [Example](#)

func [AppendFloat](#)

```
func AppendFloat(dst []byte, f float64, fmt byte, prec, bitSize int) []byte
```

AppendFloat appends the string form of the floating-point number f, as generated by FormatFloat, to dst and returns the extended buffer.

► [Example](#)

func [AppendInt](#)

```
func AppendInt(dst []byte, i int64, base int) []byte
```

AppendInt appends the string form of the integer i, as generated by FormatInt, to dst and returns the extended buffer.

► [Example](#)

func AppendQuote

```
func AppendQuote(dst []byte, s string) []byte
```

AppendQuote appends a double-quoted Go string literal representing s, as generated by Quote, to dst and returns the extended buffer.

► [Example](#)

func AppendQuoteRune

```
func AppendQuoteRune(dst []byte, r rune) []byte
```

AppendQuoteRune appends a single-quoted Go character literal representing the rune, as generated by QuoteRune, to dst and returns the extended buffer.

► [Example](#)

func AppendQuoteRuneToASCII

```
func AppendQuoteRuneToASCII(dst []byte, r rune) []byte
```

AppendQuoteRuneToASCII appends a single-quoted Go character literal representing the rune, as generated by QuoteRuneToASCII, to dst and returns the extended buffer.

► [Example](#)

func AppendQuoteRuneToGraphic

added in go1.6

```
func AppendQuoteRuneToGraphic(dst []byte, r rune) []byte
```

AppendQuoteRuneToGraphic appends a single-quoted Go character literal representing the rune, as generated by QuoteRuneToGraphic, to dst and returns the extended buffer.

func AppendQuoteToASCII

```
func AppendQuoteToASCII(dst []byte, s string) []byte
```

AppendQuoteToASCII appends a double-quoted Go string literal representing s, as generated by QuoteToASCII, to dst and returns the extended buffer.

► [Example](#)

func AppendQuoteToGraphic

added in go1.6

```
func AppendQuoteToGraphic(dst []byte, s string) []byte
```

AppendQuoteToGraphic appends a double-quoted Go string literal representing s, as generated by QuoteToGraphic, to dst and returns the extended buffer.

func AppendUint

```
func AppendUint(dst []byte, i uint64, base int) []byte
```

AppendUint appends the string form of the unsigned integer i, as generated by FormatUint, to dst and returns the extended buffer.

► [Example](#)

func Atoi

```
func Atoi(s string) (int, error)
```

Atoi is equivalent to ParseInt(s, 10, 0), converted to type int.

► [Example](#)

func CanBackquote

```
func CanBackquote(s string) bool
```

CanBackquote reports whether the string s can be represented unchanged as a single-line backquoted string without control characters other than tab.

► [Example](#)

func FormatBool

```
func FormatBool(b bool) string
```

FormatBool returns "true" or "false" according to the value of b.

► [Example](#)

```
func FormatComplex(c complex128, fmt byte, prec, bitSize int) string
```

FormatComplex converts the complex number *c* to a string of the form (a+bi) where *a* and *b* are the real and imaginary parts, formatted according to the format *fmt* and precision *prec*.

The format *fmt* and precision *prec* have the same meaning as in FormatFloat. It rounds the result assuming that the original was obtained from a complex value of *bitSize* bits, which must be 64 for complex64 and 128 for complex128.

func FormatFloat

```
func FormatFloat(f float64, fmt byte, prec, bitSize int) string
```

FormatFloat converts the floating-point number *f* to a string, according to the format *fmt* and precision *prec*. It rounds the result assuming that the original was obtained from a floating-point value of *bitSize* bits (32 for float32, 64 for float64).

The format *fmt* is one of 'b' (-ddddd±ddd, a binary exponent), 'e' (-d.dddde±dd, a decimal exponent), 'E' (-d.ddddE±dd, a decimal exponent), 'f' (-ddd.dddd, no exponent), 'g' ('e' for large exponents, 'f' otherwise), 'G' ('E' for large exponents, 'f' otherwise), 'x' (-0xd.ddddd±ddd, a hexadecimal fraction and binary exponent), or 'X' (-0Xd.dddddP±ddd, a hexadecimal fraction and binary exponent).

The precision *prec* controls the number of digits (excluding the exponent) printed by the 'e', 'E', 'f', 'g', 'G', 'x', and 'X' formats. For 'e', 'E', 'f', 'x', and 'X', it is the number of digits after the decimal point. For 'g' and 'G' it is the maximum number of significant digits (trailing zeros are removed). The special precision -1 uses the smallest number of digits necessary such that ParseFloat will return *f* exactly.

► Example

func FormatInt

```
func FormatInt(i int64, base int) string
```

FormatInt returns the string representation of *i* in the given base, for $2 \leq \text{base} \leq 36$. The result uses the lower-case letters 'a' to 'z' for digit values ≥ 10 .

► Example

func FormatUint

```
func FormatUint(i uint64, base int) string
```

FormatUint returns the string representation of *i* in the given base, for $2 \leq \text{base} \leq 36$. The result uses the lower-case letters 'a' to 'z' for digit values ≥ 10 .

► [Example](#)

func IsGraphic

added in go1.6

```
func IsGraphic(r rune) bool
```

IsGraphic reports whether the rune is defined as a Graphic by Unicode. Such characters include letters, marks, numbers, punctuation, symbols, and spaces, from categories L, M, N, P, S, and Zs.

► [Example](#)

func IsPrint

```
func IsPrint(r rune) bool
```

IsPrint reports whether the rune is defined as printable by Go, with the same definition as `unicode.IsPrint`: letters, numbers, punctuation, symbols and ASCII space.

► [Example](#)

func Itoa

```
func Itoa(i int) string
```

Itoa is equivalent to `FormatInt(int64(i), 10)`.

► [Example](#)

func ParseBool

```
func ParseBool(str string) (bool, error)
```

ParseBool returns the boolean value represented by the string. It accepts 1, t, T, TRUE, true, True, 0, f, F, FALSE, false, False. Any other value returns an error.

► [Example](#)

func ParseComplex

added in go1.15

```
func ParseComplex(s string, bitSize int) (complex128, error)
```

ParseComplex converts the string `s` to a complex number with the precision specified by `bitSize`: 64 for `complex64`, or 128 for `complex128`. When `bitSize=64`, the result still has type `complex128`, but it will be

convertible to complex64 without changing its value.

The number represented by *s* must be of the form *N*, *Ni*, or *N±Ni*, where *N* stands for a floating-point number as recognized by `parseFloat`, and *i* is the imaginary component. If the second *N* is unsigned, a + sign is required between the two components as indicated by the \pm . If the second *N* is NaN, only a + sign is accepted. The form may be parenthesized and cannot contain any spaces. The resulting complex number consists of the two components converted by `parseFloat`.

The errors that `ParseComplex` returns have concrete type `*NumError` and include `err.Num = s`.

If *s* is not syntactically well-formed, `ParseComplex` returns `err.Err = ErrSyntax`.

If *s* is syntactically well-formed but either component is more than 1/2 ULP away from the largest floating point number of the given component's size, `ParseComplex` returns `err.Err = ErrRange` and `c = ±Inf` for the respective component.

func `parseFloat`

```
func parseFloat(s string, bitSize int) (float64, error)
```

`parseFloat` converts the string *s* to a floating-point number with the precision specified by `bitSize`: 32 for `float32`, or 64 for `float64`. When `bitSize=32`, the result still has type `float64`, but it will be convertible to `float32` without changing its value.

`parseFloat` accepts decimal and hexadecimal floating-point numbers as defined by the Go syntax for [floating-point literals](#). If *s* is well-formed and near a valid floating-point number, `parseFloat` returns the nearest floating-point number rounded using IEEE754 unbiased rounding. (Parsing a hexadecimal floating-point value only rounds when there are more bits in the hexadecimal representation than will fit in the mantissa.)

The errors that `parseFloat` returns have concrete type `*NumError` and include `err.Num = s`.

If *s* is not syntactically well-formed, `parseFloat` returns `err.Err = ErrSyntax`.

If *s* is syntactically well-formed but is more than 1/2 ULP away from the largest floating point number of the given size, `parseFloat` returns `f = ±Inf`, `err.Err = ErrRange`.

`parseFloat` recognizes the string "NaN", and the (possibly signed) strings "Inf" and "Infinity" as their respective special floating point values. It ignores case when matching.

► Example

func `parseInt`

```
func parseInt(s string, base int, bitSize int) (i int64, err error)
```

`parseInt` interprets a string *s* in the given base (0, 2 to 36) and bit size (0 to 64) and returns the corresponding value *i*.

The string may begin with a leading sign: "+" or "-".

If the base argument is 0, the true base is implied by the string's prefix following the sign (if present): 2 for "0b", 8 for "0" or "0o", 16 for "0x", and 10 otherwise. Also, for argument base 0 only, underscore characters are permitted as defined by the Go syntax for [integer literals](#).

The bitSize argument specifies the integer type that the result must fit into. Bit sizes 0, 8, 16, 32, and 64 correspond to int, int8, int16, int32, and int64. If bitSize is below 0 or above 64, an error is returned.

The errors that ParseInt returns have concrete type *NumError and include err.Num = s. If s is empty or contains invalid digits, err.Err = ErrSyntax and the returned value is 0; if the value corresponding to s cannot be represented by a signed integer of the given size, err.Err = ErrRange and the returned value is the maximum magnitude integer of the appropriate bitSize and sign.

► [Example](#)

func ParseUint

```
func ParseUint(s string, base int, bitSize int) (uint64, error)
```

ParseUint is like ParseInt but for unsigned numbers.

A sign prefix is not permitted.

► [Example](#)

func Quote

```
func Quote(s string) string
```

Quote returns a double-quoted Go string literal representing s. The returned string uses Go escape sequences (\t, \n, \xFF, \u0100) for control characters and non-printable characters as defined by IsPrint.

► [Example](#)

func QuoteRune

```
func QuoteRune(r rune) string
```

QuoteRune returns a single-quoted Go character literal representing the rune. The returned string uses Go escape sequences (\t, \n, \xFF, \u0100) for control characters and non-printable characters as defined by IsPrint. If r is not a valid Unicode code point, it is interpreted as the Unicode replacement character U+FFFD.

► [Example](#)

func QuoteRuneToASCII

```
func QuoteRuneToASCII(r rune) string
```

QuoteRuneToASCII returns a single-quoted Go character literal representing the rune. The returned string uses Go escape sequences (`\t`, `\n`, `\xFF`, `\u0100`) for non-ASCII characters and non-printable characters as defined by `IsPrint`. If `r` is not a valid Unicode code point, it is interpreted as the Unicode replacement character `U+FFFD`.

► [Example](#)

func QuoteRuneToGraphic

added in go1.6

```
func QuoteRuneToGraphic(r rune) string
```

QuoteRuneToGraphic returns a single-quoted Go character literal representing the rune. If the rune is not a Unicode graphic character, as defined by `IsGraphic`, the returned string will use a Go escape sequence (`\t`, `\n`, `\xFF`, `\u0100`). If `r` is not a valid Unicode code point, it is interpreted as the Unicode replacement character `U+FFFD`.

► [Example](#)

func QuoteToASCII

```
func QuoteToASCII(s string) string
```

QuoteToASCII returns a double-quoted Go string literal representing `s`. The returned string uses Go escape sequences (`\t`, `\n`, `\xFF`, `\u0100`) for non-ASCII characters and non-printable characters as defined by `IsPrint`.

► [Example](#)

func QuoteToGraphic

added in go1.6

```
func QuoteToGraphic(s string) string
```

QuoteToGraphic returns a double-quoted Go string literal representing `s`. The returned string leaves Unicode graphic characters, as defined by `IsGraphic`, unchanged and uses Go escape sequences (`\t`, `\n`, `\xFF`, `\u0100`) for non-graphic characters.

► [Example](#)

func QuotedPrefix

added in go1.17

```
func QuotedPrefix(s string) (string, error)
```

QuotedPrefix returns the quoted string (as understood by Unquote) at the prefix of s. If s does not start with a valid quoted string, QuotedPrefix returns an error.

func Unquote

```
func Unquote(s string) (string, error)
```

Unquote interprets s as a single-quoted, double-quoted, or backquoted Go string literal, returning the string value that s quotes. (If s is single-quoted, it would be a Go character literal; Unquote returns the corresponding one-character string.)

► [Example](#)

func UnquoteChar

```
func UnquoteChar(s string, quote byte) (value rune, multibyte bool, tail string, err error)
```

UnquoteChar decodes the first character or byte in the escaped string or character literal represented by the string s. It returns four values:

1. value, the decoded Unicode code point or byte value;
2. multibyte, a boolean indicating whether the decoded character requires a multibyte UTF-8 representation;
3. tail, the remainder of the string after the character; and
4. an error that will be nil if the character is syntactically valid.

The second argument, quote, specifies the type of literal being parsed and therefore which escaped quote character is permitted. If set to a single quote, it permits the sequence `\'` and disallows unescaped `'`. If set to a double quote, it permits `\"` and disallows unescaped `"`. If set to zero, it does not permit either escape and allows both quote characters to appear unescaped.

► [Example](#)

Types

type NumError

```
type NumError struct {  
    Func string // the failing function (ParseBool, ParseInt, ParseUint, ParseFloat, ParseInt64, ParseInt32, ParseInt16, ParseInt8, ParseInt4, ParseInt2, ParseInt1, ParseInt0, ParseFloat64, ParseFloat32, ParseFloat16, ParseFloat8, ParseFloat4, ParseFloat2, ParseFloat1, ParseFloat0)  
    Num  string // the input  
    Err  error  // the reason the conversion failed (e.g. ErrRange, ErrSyntax, etc.)  
}
```

A NumError records a failed conversion.

func (*NumError) **Error**

```
func (e *NumError) Error() string
```

func (*NumError) **Unwrap**

added in go1.14

```
func (e *NumError) Unwrap() error
```



Source Files

[View all](#)

[atob.go](#)
[atoc.go](#)
[atof.go](#)
[atoi.go](#)
[bytealg.go](#)

[ctoa.go](#)
[decimal.go](#)
[doc.go](#)
[eisel_lemire.go](#)
[ftoa.go](#)

[ftoaryu.go](#)
[isprint.go](#)
[itoa.go](#)
[quote.go](#)

Why Go

[Use Cases](#)
[Case Studies](#)

Get Started

[Playground](#)
[Tour](#)
[Stack Overflow](#)
[Help](#)

Packages

[Standard Library](#)
[About Go Packages](#)

About

[Download](#)
[Blog](#)
[Issue Tracker](#)
[Release Notes](#)
[Brand Guidelines](#)
[Code of Conduct](#)

Connect

[Twitter](#)
[GitHub](#)
[Slack](#)
[r/golang](#)
[Meetup](#)
[Golang Weekly](#)



Report an Issue

Google

