fmt

Package fmt go1.15.2 Latest

Published: **Sep 9, 2020** | License: BSD-3-Clause | Standard library

Overview

Package fmt implements formatted I/O with functions analogous to C's printf and scanf. The format 'verbs' are derived from C's but are simpler.

Printing

The verbs:

General:

```
%v  the value in a default format
    when printing structs, the plus flag (%+v) adds field names
%#v a Go-syntax representation of the value
%T  a Go-syntax representation of the type of the value
%%  a literal percent sign; consumes no value
```

Boolean:

```
%t  the word true or false
```

Integer:

```
%b  base 2
%c  the character represented by the corresponding Unicode code point
%d  base 10
%o  base 8
%O  base 8 with 0o prefix
%q  a single-quoted character literal safely escaped with Go syntax.
%x  base 16, with lower-case letters for a-f
%X  base 16, with upper-case letters for A-F
%U  Unicode format: U+1234; same as "U+%04X"
```

Floating-point and complex constituents:

```
%b  decimalless scientific notation with exponent a power of two,
    in the manner of strconv.FormatFloat with the 'b' format,
    e.g. -123456p-78
%e  scientific notation, e.g. -1.234456e+78
%E  scientific notation, e.g. -1.234456E+78
```

```
%f  decimal point but no exponent, e.g. 123.456
%F  synonym for %f
%g  %e for large exponents, %f otherwise. Precision is discussed below.
%G  %E for large exponents, %F otherwise
%x  hexadecimal notation (with decimal power of two exponent), e.g. -0x1.23abcp+20
%X  upper-case hexadecimal notation, e.g. -0X1.23ABCP+20
```

String and slice of bytes (treated equivalently with these verbs):

```
%s  the uninterpreted bytes of the string or slice
%q  a double-quoted string safely escaped with Go syntax
%x  base 16, lower-case, two characters per byte
%X  base 16, upper-case, two characters per byte
```

Slice:

```
%p  address of 0th element in base 16 notation, with leading 0x
```

Pointer:

```
%p  base 16 notation, with leading 0x
The %b, %d, %o, %x and %X verbs also work with pointers,
formatting the value exactly as if it were an integer.
```

The default format for %v is:

```
bool:                    %t
int, int8 etc.:          %d
uint, uint8 etc.:        %d, %#x if printed with %#v
float32, complex64, etc: %g
string:                  %s
chan:                    %p
pointer:                 %p
```

For compound objects, the elements are printed using these rules, recursively, laid out like this:

```
struct:             {field0 field1 ...}
array, slice:       [elem0 elem1 ...]
maps:               map[key1:value1 key2:value2 ...]
pointer to above:   &{}, &[], &map[]
```

Width is specified by an optional decimal number immediately preceding the verb. If absent, the width is whatever is necessary to represent the value. Precision is specified after the (optional) width by a period followed by a decimal number. If no period is present, a default precision is used. A period with no following number specifies a precision of zero. Examples:

```
%f     default width, default precision
%9f    width 9, default precision
```

```
%.2f   default width, precision 2
%9.2f  width 9, precision 2
%9.f   width 9, precision 0
```

Width and precision are measured in units of Unicode code points, that is, runes. (This differs from C's printf where the units are always measured in bytes.) Either or both of the flags may be replaced with the character '*', causing their values to be obtained from the next operand (preceding the one to format), which must be of type int.

For most values, width is the minimum number of runes to output, padding the formatted form with spaces if necessary.

For strings, byte slices and byte arrays, however, precision limits the length of the input to be formatted (not the size of the output), truncating if necessary. Normally it is measured in runes, but for these types when formatted with the %x or %X format it is measured in bytes.

For floating-point values, width sets the minimum width of the field and precision sets the number of places after the decimal, if appropriate, except that for %g/%G precision sets the maximum number of significant digits (trailing zeros are removed). For example, given 12.345 the format %6.3f prints 12.345 while %.3g prints 12.3. The default precision for %e, %f and %#g is 6; for %g it is the smallest number of digits necessary to identify the value uniquely.

For complex numbers, the width and precision apply to the two components independently and the result is parenthesized, so %f applied to 1.2+3.4i produces (1.200000+3.400000i).

Other flags:

```
+   always print a sign for numeric values;
    guarantee ASCII-only output for %q (%+q)
-   pad with spaces on the right rather than the left (left-justify the field)
#   alternate format: add leading 0b for binary (%#b), 0 for octal (%#o),
    0x or 0X for hex (%#x or %#X); suppress 0x for %p (%#p);
    for %q, print a raw (backquoted) string if strconv.CanBackquote
    returns true;
    always print a decimal point for %e, %E, %f, %F, %g and %G;
    do not remove trailing zeros for %g and %G;
    write e.g. U+0078 'x' if the character is printable for %U (%#U).
' ' (space) leave a space for elided sign in numbers (% d);
    put spaces between bytes printing strings or slices in hex (% x, % X)
0   pad with leading zeros rather than spaces;
    for numbers, this moves the padding after the sign
```

Flags are ignored by verbs that do not expect them. For example there is no alternate decimal format, so %#d and %d behave identically.

For each Printf-like function, there is also a Print function that takes no format and is equivalent to saying %v for every operand. Another variant Println inserts blanks between operands and appends a newline.

Regardless of the verb, if an operand is an interface value, the internal concrete value is used, not the interface itself. Thus:

```
var i interface{} = 23
fmt.Printf("%v\n", i)
```

will print 23.

Except when printed using the verbs %T and %p, special formatting considerations apply for operands that implement certain interfaces. In order of application:

1. If the operand is a reflect.Value, the operand is replaced by the concrete value that it holds, and printing continues with the next rule.

2. If an operand implements the Formatter interface, it will be invoked. Formatter provides fine control of formatting.

3. If the %v verb is used with the # flag (%#v) and the operand implements the GoStringer interface, that will be invoked.

If the format (which is implicitly %v for Println etc.) is valid for a string (%s %q %v %x %X), the following two rules apply:

4. If an operand implements the error interface, the Error method will be invoked to convert the object to a string, which will then be formatted as required by the verb (if any).

5. If an operand implements method String() string, that method will be invoked to convert the object to a string, which will then be formatted as required by the verb (if any).

For compound operands such as slices and structs, the format applies to the elements of each operand, recursively, not to the operand as a whole. Thus %q will quote each element of a slice of strings, and %6.2f will control formatting for each element of a floating-point array.

However, when printing a byte slice with a string-like verb (%s %q %x %X), it is treated identically to a string, as a single item.

To avoid recursion in cases such as

```
type X string
func (x X) String() string { return Sprintf("<%s>", x) }
```

convert the value before recurring:

```
func (x X) String() string { return Sprintf("<%s>", string(x)) }
```

Infinite recursion can also be triggered by self-referential data structures, such as a slice that contains itself as an element, if that type has a String method. Such pathologies are rare, however, and the package does not protect against them.

When printing a struct, fmt cannot and therefore does not invoke formatting methods such as Error or String on unexported fields.

Explicit argument indexes:

In Printf, Sprintf, and Fprintf, the default behavior is for each formatting verb to format successive arguments passed in the call. However, the notation [n] immediately before the verb indicates that the nth one-indexed argument is to be formatted instead. The same notation before a '*' for a width or precision selects the argument index holding the value. After processing a bracketed expression [n], subsequent verbs will use arguments n+1, n+2, etc. unless otherwise directed.

For example,

```
fmt.Sprintf("%[2]d %[1]d\n", 11, 22)
```

will yield "22 11", while

```
fmt.Sprintf("%[3]*.[2]*[1]f", 12.0, 2, 6)
```

equivalent to

```
fmt.Sprintf("%6.2f", 12.0)
```

will yield " 12.00". Because an explicit index affects subsequent verbs, this notation can be used to print the same values multiple times by resetting the index for the first argument to be repeated:

```
fmt.Sprintf("%d %d %#[1]x %#x", 16, 17)
```

will yield "16 17 0x10 0x11".

Format errors:

If an invalid argument is given for a verb, such as providing a string to %d, the generated string will contain a description of the problem, as in these examples:

```
Wrong type or unknown verb: %!verb(type=value)
    Printf("%d", "hi"):        %!d(string=hi)
Too many arguments: %!(EXTRA type=value)
    Printf("hi", "guys"):      hi%!(EXTRA string=guys)
Too few arguments: %!verb(MISSING)
    Printf("hi%d"):            hi%!d(MISSING)
Non-int for width or precision: %!(BADWIDTH) or %!(BADPREC)
    Printf("%*s", 4.5, "hi"):  %!(BADWIDTH)hi
    Printf("%.*s", 4.5, "hi"): %!(BADPREC)hi
Invalid or invalid use of argument index: %!(BADINDEX)
    Printf("%*[2]d", 7):       %!d(BADINDEX)
    Printf("%.[2]d", 7):       %!d(BADINDEX)
```

All errors begin with the string "%!" followed sometimes by a single character (the verb) and end with a parenthesized description.

If an Error or String method triggers a panic when called by a print routine, the fmt package reformats the error message from the panic, decorating it with an indication that it came through the fmt package. For example, if a String method calls panic("bad"), the resulting formatted message will look like

```
%!s(PANIC=bad)
```

The %!s just shows the print verb in use when the failure occurred. If the panic is caused by a nil receiver to an Error or String method, however, the output is the undecorated string, "<nil>".

Scanning

An analogous set of functions scans formatted text to yield values. Scan, Scanf and Scanln read from os.Stdin; Fscan, Fscanf and Fscanln read from a specified io.Reader; Sscan, Sscanf and Sscanln read from an argument string.

Scan, Fscan, Sscan treat newlines in the input as spaces.

Scanln, Fscanln and Sscanln stop scanning at a newline and require that the items be followed by a newline or EOF.

Scanf, Fscanf, and Sscanf parse the arguments according to a format string, analogous to that of Printf. In the text that follows, 'space' means any Unicode whitespace character except newline.

In the format string, a verb introduced by the % character consumes and parses input; these verbs are described in more detail below. A character other than %, space, or newline in the format consumes exactly that input character, which must be present. A newline with zero or more spaces before it in the format string consumes zero or more spaces in the input followed by a single newline or the end of the input. A space following a newline in the format string consumes zero or more spaces in the input. Otherwise, any run of one or more spaces in the format string consumes as many spaces as possible in the input. Unless the run of spaces in the format string appears adjacent to a newline, the run must consume at least one space from the input or find the end of the input.

The handling of spaces and newlines differs from that of C's scanf family: in C, newlines are treated as any other space, and it is never an error when a run of spaces in the format string finds no spaces to consume in the input.

The verbs behave analogously to those of Printf. For example, %x will scan an integer as a hexadecimal number, and %v will scan the default representation format for the value. The Printf verbs %p and %T and the flags # and + are not implemented. For floating-point and complex values, all valid formatting verbs (%b %e %E %f %F %g %G %x %X and %v) are equivalent and accept both decimal and hexadecimal notation (for example: "2.3e+7", "0x4.5p-8") and digit-separating underscores (for example: "3.14159_26535_89793").

Input processed by verbs is implicitly space-delimited: the implementation of every verb except %c starts by discarding leading spaces from the remaining input, and the %s verb (and %v reading into a string) stops consuming input at the first space or newline character.

The familiar base-setting prefixes 0b (binary), 0o and 0 (octal), and 0x (hexadecimal) are accepted when scanning integers without a format or with the %v verb, as are digit-separating underscores.

Width is interpreted in the input text but there is no syntax for scanning with a precision (no %5.2f, just %5f). If width is provided, it applies after leading spaces are trimmed and specifies the maximum number of runes to read to satisfy the verb. For example,

```
Sscanf(" 1234567 ", "%5s%d", &s, &i)
```

will set s to "12345" and i to 67 while

```
Sscanf(" 12 34 567 ", "%5s%d", &s, &i)
```

will set s to "12" and i to 34.

In all the scanning functions, a carriage return followed immediately by a newline is treated as a plain newline (\r\n means the same as \n).

In all the scanning functions, if an operand implements method Scan (that is, it implements the Scanner interface) that method will be used to scan the text for that operand. Also, if the number of arguments scanned is less than the number of arguments provided, an error is returned.

All arguments to be scanned must be either pointers to basic types or implementations of the Scanner interface.

Like Scanf and Fscanf, Sscanf need not consume its entire input. There is no way to recover how much of the input string Sscanf used.

Note: Fscan etc. can read one character (rune) past the input they return, which means that a loop calling a scan routine may skip some of the input. This is usually a problem only when there is no space between input values. If the reader provided to Fscan implements ReadRune, that method will be used to read characters. If the reader also implements UnreadRune, that method will be used to save the character and successive calls will not lose data. To attach ReadRune and UnreadRune methods to a reader without that capability, use bufio.NewReader.

func Errorf

```
func Errorf(format string, a ...interface{}) error
```

Errorf formats according to a format specifier and returns the string as a value that satisfies error.

If the format specifier includes a %w verb with an error operand, the returned error will implement an Unwrap method returning the operand. It is invalid to include more than one %w verb or to supply it

with an operand that does not implement the error interface. The %w verb is otherwise a synonym for %v.

func Fprint

```
func Fprint(w io.Writer, a ...interface{}) (n int, err error)
```

Fprint formats using the default formats for its operands and writes to w. Spaces are added between operands when neither is a string. It returns the number of bytes written and any write error encountered.

func Fprintf

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

Fprintf formats according to a format specifier and writes to w. It returns the number of bytes written and any write error encountered.

func Fprintln

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)
```

Fprintln formats using the default formats for its operands and writes to w. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.

func Fscan

```
func Fscan(r io.Reader, a ...interface{}) (n int, err error)
```

Fscan scans text read from r, storing successive space-separated values into successive arguments. Newlines count as space. It returns the number of items successfully scanned. If that is less than the number of arguments, err will report why.

func Fscanf

```
func Fscanf(r io.Reader, format string, a ...interface{}) (n int, err error)
```

Fscanf scans text read from r, storing successive space-separated values into successive arguments as determined by the format. It returns the number of items successfully parsed. Newlines in the input must match newlines in the format.

func Fscanln

```
func Fscanln(r io.Reader, a ...interface{}) (n int, err error)
```

Fscanln is similar to Fscan, but stops scanning at a newline and after the final item there must be a newline or EOF.

func Print

```
func Print(a ...interface{}) (n int, err error)
```

Print formats using the default formats for its operands and writes to standard output. Spaces are added between operands when neither is a string. It returns the number of bytes written and any write error encountered.

func Printf

```
func Printf(format string, a ...interface{}) (n int, err error)
```

Printf formats according to a format specifier and writes to standard output. It returns the number of bytes written and any write error encountered.

func Println

```
func Println(a ...interface{}) (n int, err error)
```

Println formats using the default formats for its operands and writes to standard output. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.

func Scan

```
func Scan(a ...interface{}) (n int, err error)
```

Scan scans text read from standard input, storing successive space-separated values into successive arguments. Newlines count as space. It returns the number of items successfully scanned. If that is less than the number of arguments, err will report why.

func Scanf

```
func Scanf(format string, a ...interface{}) (n int, err error)
```

Scanf scans text read from standard input, storing successive space-separated values into successive arguments as determined by the format. It returns the number of items successfully scanned. If that is less than the number of arguments, err will report why. Newlines in the input must match newlines in the format. The one exception: the verb %c always scans the next rune in the input, even if it is a space (or tab etc.) or newline.

func Scanln

```
func Scanln(a ...interface{}) (n int, err error)
```

Scanln is similar to Scan, but stops scanning at a newline and after the final item there must be a newline or EOF.

func Sprint

```
func Sprint(a ...interface{}) string
```

Sprint formats using the default formats for its operands and returns the resulting string. Spaces are added between operands when neither is a string.

func Sprintf

```
func Sprintf(format string, a ...interface{}) string
```

Sprintf formats according to a format specifier and returns the resulting string.

func Sprintln

```
func Sprintln(a ...interface{}) string
```

Sprintln formats using the default formats for its operands and returns the resulting string. Spaces are always added between operands and a newline is appended.

func Sscan

```
func Sscan(str string, a ...interface{}) (n int, err error)
```

Sscan scans the argument string, storing successive space-separated values into successive arguments. Newlines count as space. It returns the number of items successfully scanned. If that is less than the number of arguments, err will report why.

func Sscanf

```
func Sscanf(str string, format string, a ...interface{}) (n int, err error)
```

Sscanf scans the argument string, storing successive space-separated values into successive arguments as determined by the format. It returns the number of items successfully parsed. Newlines in the input must match newlines in the format.

func Sscanln

```
func Sscanln(str string, a ...interface{}) (n int, err error)
```

Sscanln is similar to Sscan, but stops scanning at a newline and after the final item there must be a newline or EOF.

type Formatter

```
type Formatter interface {
    Format(f State, c rune)
}
```

Formatter is the interface implemented by values with a custom formatter. The implementation of Format may call Sprint(f) or Fprint(f) etc. to generate its output.

type GoStringer

```
type GoStringer interface {
    GoString() string
}
```

GoStringer is implemented by any value that has a GoString method, which defines the Go syntax for that value. The GoString method is used to print values passed as an operand to a %#v format.

type ScanState

```
type ScanState interface {
    // ReadRune reads the next rune (Unicode code point) from the input.
    // If invoked during Scanln, Fscanln, or Sscanln, ReadRune() will
    // return EOF after returning the first '\n' or when reading beyond
    // the specified width.
    ReadRune() (r rune, size int, err error)
    // UnreadRune causes the next call to ReadRune to return the same rune.
    UnreadRune() error
    // SkipSpace skips space in the input. Newlines are treated appropriately
    // for the operation being performed; see the package documentation
    // for more information.
    SkipSpace()
    // Token skips space in the input if skipSpace is true, then returns the
    // run of Unicode code points c satisfying f(c).  If f is nil,
    // !unicode.IsSpace(c) is used; that is, the token will hold non-space
    // characters. Newlines are treated appropriately for the operation being
    // performed; see the package documentation for more information.
    // The returned slice points to shared data that may be overwritten
    // by the next call to Token, a call to a Scan function using the ScanState
    // as input, or when the calling Scan method returns.
    Token(skipSpace bool, f func(rune) bool) (token []byte, err error)
    // Width returns the value of the width option and whether it has been set.
    // The unit is Unicode code points.
    Width() (wid int, ok bool)
    // Because ReadRune is implemented by the interface, Read should never be
    // called by the scanning routines and a valid implementation of
    // ScanState may choose always to return an error from Read.
```

```
    Read(buf []byte) (n int, err error)
}
```

ScanState represents the scanner state passed to custom scanners. Scanners may do rune-at-a-time scanning or ask the ScanState to discover the next space-delimited token.

type Scanner

```
type Scanner interface {
    Scan(state ScanState, verb rune) error
}
```

Scanner is implemented by any value that has a Scan method, which scans the input for the representation of a value and stores the result in the receiver, which must be a pointer to be useful. The Scan method is called for any argument to Scan, Scanf, or Scanln that implements it.

type State

```
type State interface {
    // Write is the function to call to emit formatted output to be printed.
    Write(b []byte) (n int, err error)
    // Width returns the value of the width option and whether it has been set.
    Width() (wid int, ok bool)
    // Precision returns the value of the precision option and whether it has been se
    Precision() (prec int, ok bool)

    // Flag reports whether the flag c, a character, has been set.
    Flag(c int) bool
}
```

State represents the printer state passed to custom formatters. It provides access to the io.Writer interface plus information about the flags and options for the operand's format specifier.

type Stringer

```
type Stringer interface {
    String() string
}
```

Stringer is implemented by any value that has a String method, which defines the ``native" format for that value. The String method is used to print values passed as an operand to any format that accepts a string or to an unformatted printer such as Print.