Contents

The l3str-format package: formatting strings of characters[*]

The LaTeX3 Project[†]

Released 2012/07/09

1 Format specifications

In this module, we introduce the notion of a string ⟨*format*⟩. The syntax follows that of Python's `format` built-in function. A ⟨*format specification*⟩ is a string of the form

$$\langle\textit{format specification}\rangle = [[\langle\textit{fill}\rangle]\langle\textit{alignment}\rangle][\langle\textit{sign}\rangle][\langle\textit{width}\rangle][.\langle\textit{precision}\rangle][\langle\textit{style}\rangle]$$

where each [...] denotes an independent optional part.

- ⟨*fill*⟩ can be any character: it is assumed to be present whenever the second character of the ⟨*format specification*⟩ is a valid ⟨*alignment*⟩ character.
- ⟨*alignment*⟩ can be `<` (left alignment), `>` (right alignment), `^` (centering), or `=` (for numeric types only).
- ⟨*sign*⟩ is allowed for numeric types; it can be `+` (show a sign for positive and negative numbers), `-` (only put a sign for negative numbers), or a space (show a space or a `-`).
- ⟨*width*⟩ is the minimum number of characters of the result: if the result is naturally shorter than this ⟨*width*⟩, then it is padded with copies of the character ⟨*fill*⟩, with a position depending on the choice of ⟨*alignment*⟩. If the result is naturally longer, it is not truncated.
- ⟨*precision*⟩, whose presence is indicated by a period, can have different meanings depending on the type.
- ⟨*style*⟩ is one character, which controls how the given data should be formatted. The list of allowed ⟨*styles*⟩ depends on the type.

The choice of ⟨*alignment*⟩ `=` is only valid for numeric types: in this case the padding is inserted between the sign and the rest of the number.

[*] This file describes v3940, last revised 2012/07/09.

[†] E-mail: latex-team@latex-project.org

2 Formatting various data-types

`\tl_format:Nn` ★
`\tl_format:(cn|nn)` ★

`\tl_format:nn {⟨token list⟩} {⟨format specification⟩}`

Converts the ⟨*token list*⟩ to a string according to the ⟨*format specification*⟩. The ⟨*style*⟩, if present, must be `s`. If ⟨*precision*⟩ is given, all characters of the string representation of the ⟨*token list*⟩ beyond the first ⟨*precision*⟩ characters are discarded.

`\seq_format:Nn` ★
`\seq_format:cn` ★

`\seq_format:Nn {⟨sequence⟩} {⟨format specification⟩}`

Converts each item in the ⟨*sequence*⟩ to a string according to the ⟨*format specification*⟩, and concatenates the results.

`\int_format:nn` ★

`\int_format:nn {⟨intexpr⟩} {⟨format specification⟩}`

Evaluates the ⟨*integer expression*⟩ and converts the result to a string according to the ⟨*format specification*⟩. The ⟨*precision*⟩ argument is not allowed. The ⟨*style*⟩ can be `b` for binary output, `d` for decimal output (this is the default), `o` for octal output, `X` for hexadecimal output (using capital letters).

`\fp_format:nn` ★

`\fp_format:nn {⟨fpexpr⟩} {⟨format specification⟩}`

Evaluates the ⟨*floating point expression*⟩ and converts the result to a string according to the ⟨*format specification*⟩. The ⟨*precision*⟩ defaults to 6. The ⟨*style*⟩ can be

- `e` for scientific notation, with one digit before and ⟨*precision*⟩ digits after the decimal separator, and an integer exponent, following `e`;
- `f` for a fixed point notation, with ⟨*precision*⟩ digits after the decimal separator and no exponent;
- `g` for a general format, which uses style `f` for numbers in the range $[10^{-4}, 10^{\langle precision\rangle})$ and style `e` otherwise.

3 Possibilities, and things to do

- Provide a token list formatting ⟨*style*⟩ which keeps the last ⟨*precision*⟩ characters rather than the first ⟨*precision*⟩.

4 l3str-format implementation

```
⟨*initex | package⟩
⟨@@=str_format⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

```
\RequirePackage{l3str}
⟨/package⟩
```

4.1 Helpers

\use:nf
\use:fnf

A simple variant.

```
\cs_generate_variant:Nn \use:nn { nf }
\cs_generate_variant:Nn \use:nnn { fnf }
```

(*End definition for* `\use:nf` *and* `\use:fnf`*.*)

\tl_to_str:f

A simple variant.

```
\cs_generate_variant:Nn \tl_to_str:n { f }
```

(*End definition for* `\tl_to_str:f`*.*)

__str_format_if_digit:NTF

Here we expect `#1` to be a character with category other, or `\s__stop`.

```
\prg_new_conditional:Npnn \__str_format_if_digit:N #1 { TF }
  {
    \if_int_compare:w \c_nine < 1 #1 \exp_stop_f:
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\__str_format_if_digit:NTF`*.*)

__str_format_put:nw
__str_format_put:ow
__str_format_put:fw

Put `#1` after an `\s__stop` delimiter.

```
\cs_new:Npn \__str_format_put:nw #1 #2 \s__stop { #2 \s__stop #1 }
\cs_generate_variant:Nn \__str_format_put:nw { o , f }
```

(*End definition for* `\__str_format_put:nw`*,* `\__str_format_put:ow`*, and* `\__str_format_put:fw`*.*)

__str_format_if_in:nN*TF*
__str_format_if_in_aux:NN

A copy of `\__str_if_contains_char:nNTF` to avoid relying on this weird internal string function.

```
\prg_new_conditional:Npnn \__str_format_if_in:nN #1#2 { TF }
  {
    \__str_format_if_in_aux:NN #2 #1
      { #2 \prg_return_false: \exp_after:wN \__prg_break: \else: }
    \__prg_break_point:
  }
\cs_new:Npn \__str_format_if_in_aux:NN #1#2
  {
    \if_charcode:w #1 #2
      \prg_return_true:
      \exp_after:wN \__prg_break:
    \fi:
    \__str_format_if_in_aux:NN #1
  }
```

(*End definition for* `\__str_format_if_in:nN`*. This function is documented on page* **??**.)

4.2 Parsing a format specification

__str_format_parse:n
__str_format_parse_auxi:NN
__str_format_parse_auxii:nN
__str_format_parse_auxiii:nN
__str_format_parse_auxiv:nwN
__str_format_parse_auxv:nN
__str_format_parse_auxvi:nwN
__str_format_parse_auxvii:nN
__str_format_parse_end:nwn

The goal is to parse

⟨*format specification*⟩ = [[⟨*fill*⟩]⟨*alignment*⟩][⟨*sign*⟩][⟨*width*⟩][.⟨*precision*⟩][⟨*style*⟩]

```
\cs_new:Npn \__str_format_parse:n #1
  {
    \exp_last_unbraced:Nf \__str_format_parse_auxi:NN
      \__str_to_other:n {#1} \s__stop \s__stop {#1}
  }
\cs_new:Npx \__str_format_parse_auxi:NN #1#2
  {
    \exp_not:N \__str_format_if_in:nNTF { < > = ^ } #2
      { \exp_not:N \__str_format_parse_auxiii:nN { #1 #2 } }
      {
        \exp_not:N \__str_format_parse_auxii:nN
          { \c_catcode_other_space_tl } #1 #2
      }
  }
\cs_new:Npn \__str_format_parse_auxii:nN #1#2
  {
    \__str_format_if_in:nNTF { < > = ^ } #2
      { \__str_format_parse_auxiii:nN { #1 #2 } }
      { \__str_format_parse_auxiii:nN { #1 ? } #2 }
  }
\cs_new:Npx \__str_format_parse_auxiii:nN #1#2
  {
    \exp_not:N \__str_format_if_in:nNTF
      { + - \c_catcode_other_space_tl }
      #2
      { \exp_not:N \__str_format_parse_auxiv:nwN { #1 #2 } ; }
      { \exp_not:N \__str_format_parse_auxiv:nwN { #1 ? } ; #2 }
  }
\cs_new:Npn \__str_format_parse_auxiv:nwN #1#2; #3
  {
    \__str_format_if_digit:NTF #3
      { \__str_format_parse_auxiv:nwN {#1} #2 #3 ; }
      { \__str_format_parse_auxv:nN { #1 {#2} } #3 }
  }
\cs_new:Npn \__str_format_parse_auxv:nN #1#2
  {
    \token_if_eq_charcode:NNTF . #2
      { \__str_format_parse_auxvi:nwN {#1} 0 ; }
      { \__str_format_parse_auxvii:nN { #1 { } } #2 }
  }
\cs_new:Npn \__str_format_parse_auxvi:nwN #1#2; #3
  {
    \__str_format_if_digit:NTF #3
      { \__str_format_parse_auxvi:nwN {#1} #2 #3 ; }
      { \__str_format_parse_auxvii:nN { #1 {#2} } #3 }
  }
\cs_new:Npn \__str_format_parse_auxvii:nN #1#2
  {
    \token_if_eq_meaning:NNTF \s__stop #2
      { \__str_format_parse_end:nwn { #1 ? } #2 }
      { \__str_format_parse_end:nwn { #1 #2 } }
  }
\cs_new:Npn \__str_format_parse_end:nwn #1 #2 \s__stop \s__stop #3
  {
    \tl_if_empty:nF {#2}
      { \__msg_kernel_expandable_error:nnn { str } { invalid-format } {#3} }
    #1
  }
```

(*End definition for* `\__str_format_parse:n`. *This function is documented on page* **??**.)

4.3 Alignment

The 4 functions in this section receive an ⟨*body*⟩, a ⟨*sign*⟩, a ⟨*width*⟩ and a ⟨*fill*⟩ character (exactly one character). For non-numeric types, the ⟨*sign*⟩ is empty and the ⟨*body*⟩ is the (other) string we want to format. For numeric types, we wish to format ⟨*sign*⟩ ⟨*body*⟩ (both are other strings). The alignment types `<`, `>` and `^` keep ⟨*sign*⟩ and ⟨*body*⟩ together. The `=` alignment type, however, inserts the padding between the ⟨*sign*⟩ and the ⟨*body*⟩, hence the need to keep those separate.

__str_format_align_<:nnnN

`\__str_format_align_<:nnnN {`⟨*body*⟩`} {`⟨*sign*⟩`} {`⟨*width*⟩`}` ⟨*fill*⟩

Aligning "⟨*sign*⟩ ⟨*body*⟩" to the left entails appending `#4` the correct number of times. Then convert the result to a string.

```
\cs_new:cpn { __str_format_align_<:nnnN } #1#2#3#4
  {
    \use:nf { #2 #1 }
      {
        \prg_replicate:nn
          { \int_max:nn { #3 - \__str_count_unsafe:n { #2 #1 } } { 0 } }
          {#4}
      }
  }
```

(*End definition for* `\__str_format_align_<:nnnN`.)

__str_format_align_>:nnnN

`\__str_format_align_>:nnnN {`⟨*body*⟩`} {`⟨*sign*⟩`} {`⟨*width*⟩`}` ⟨*fill*⟩

Aligning an "⟨*sign*⟩ ⟨*body*⟩" to the right entails prepending `#4` the correct number of times. Then convert the result to a string.

```
\cs_new:cpn { __str_format_align_>:nnnN } #1#2#3#4
  {
    \prg_replicate:nn
      { \int_max:nn { #3 - \__str_count_unsafe:n { #2 #1 } } { 0 } }
      {#4}
    #2 #1
  }
```

(*End definition for* __str_format_align_>:nnnN.)

__str_format_align_^:nnnN

__str_format_align_^:nnnN {⟨*body*⟩} {⟨*sign*⟩} {⟨*width*⟩} ⟨*fill*⟩

Centering "⟨*sign*⟩ ⟨*body*⟩" entails prepending and appending #4 the correct number of times. If the number of #4 to be added is odd, we add one more after than before.

```
\cs_new:cpn { __str_format_align_^:nnnN } #1#2#3#4
  {
    \use:fnf
      {
        \prg_replicate:nn
          {
            \int_max:nn \c_zero
              { #3 - \__str_count_unsafe:n { #2 #1 } - \c_one }
            / \c_two
          }
          {#4}
      }
      { #2 #1 }
      {
        \prg_replicate:nn
          {
            \int_max:nn \c_zero
              { #3 - \__str_count_unsafe:n { #2 #1 } }
            / \c_two
          }
          {#4}
      }
  }
```

__str_format_align_=:nnnN

__str_format_align_=:nnnN {⟨*body*⟩} {⟨*sign*⟩} {⟨*width*⟩} ⟨*fill*⟩

The special numeric alignment = means that we insert the appropriate number of copies of #4 between the ⟨*sign*⟩ and the ⟨*body*⟩. Then convert the result to a string.

```
\cs_new:cpn { __str_format_align_=:nnnN } #1#2#3#4
  {
    \use:nf {#2}
      {
        \prg_replicate:nn
          { \int_max:nn { #3 - \__str_count_unsafe:n { #2 #1 } } { 0 } }
          {#4}
      }
    #1
  }
```

(*End definition for* __str_format_align_=:nnnN.)

4.4 Formatting token lists

\tl_format:Nn
\tl_format:cn
\tl_format:nn

Call __str_format_tl:NNNnnNn to read the parsed ⟨*format specification*⟩. Then convert the result to a string.

```
\cs_new_nopar:Npn \tl_format:Nn { \exp_args:No \tl_format:nn }
\cs_generate_variant:Nn \tl_format:Nn { c }
\cs_new:Npn \tl_format:nn #1#2
  {
    \tl_to_str:f
      {
        \exp_last_unbraced:Nf \__str_format_tl:NNNnnNn
          { \__str_format_parse:n {#2} }
          {#1}
      }
  }
```

(*End definition for* `\tl_format:Nn`*,* `\tl_format:cn`*, and* `\tl_format:nn`*. These functions are documented on page* **??**.)

`\__str_format_tl:NNNnnNn`

`\__str_format_tl:NNNnnNn` ⟨*fill*⟩ ⟨*alignment*⟩ ⟨*sign*⟩ `{`⟨*width*⟩`}` `{`⟨*precision*⟩`}` ⟨*style*⟩ `{`⟨*token list*⟩`}`

First check that the ⟨*alignment*⟩ is not `=`, and set the default alignment `?` to `<`. Place the modified information after a trailing `\s__stop` for later retrieval. Then check that there was no ⟨*sign*⟩. The width will be useful later, store it after `\s__stop`. Afterwards, store the precision, and the function `\__str_range_unsafe:nnn` that will be used to extract the first `#5` characters of the string. There is a need to use the "unsafe" function, as otherwise leading spaces would get stripped by `f`-expansion. Finally, check that the ⟨*style*⟩ is `?` or `s`.

```
\cs_new:Npn \__str_format_tl:NNNnnNn #1#2#3#4#5#6
  {
    \token_if_eq_charcode:NNTF #2 =
      {
        \__msg_kernel_expandable_error:nnnn
          { str } { invalid-align-format } {#2} {tl}
        \__str_format_put:nw { #1 < }
      }
      {
        \token_if_eq_charcode:NNTF #2 ?
          { \__str_format_put:nw { #1 < } }
          { \__str_format_put:nw { #1 #2 } }
      }
    \token_if_eq_charcode:NNF #3 ?
      {
        \__msg_kernel_expandable_error:nnnn
          { str } { invalid-sign-format } {#3} {tl}
      }
    \__str_format_put:nw { {#4} }
    \tl_if_empty:nTF {#5}
      { \__str_format_put:nw { \__str_range_unsafe:nnn { {1} {-1} } } }
      { \__str_format_put:nw { \__str_range_unsafe:nnn { {1} {#5} } } }
    \token_if_eq_charcode:NNF #6 s
      {
        \token_if_eq_charcode:NNF #6 ?
          {
```

```
      \__msg_kernel_expandable_error:nnnn
       { str } { invalid-style-format } {#6} {tl}
     }
   }
  \__str_format_tl_s:NNnnNNn
  \s__stop
 }
```

(*End definition for* `\__str_format_tl:NNNnnNn`.)

`\__str_format_tl_s:NNnnNNn`

`\__str_format_tl_s:NNnnNNn \s__stop` ⟨*function*⟩ `{`⟨*arguments*⟩`}` `{`⟨*width*⟩`}` ⟨*fill*⟩ ⟨*alignment*⟩ `{`⟨*token list*⟩`}`

The ⟨*function*⟩ and ⟨*arguments*⟩ are built in such a way that `f`-expanding ⟨*function*⟩ `{`⟨*other string*⟩`}` ⟨*arguments*⟩ yields the piece of the ⟨*other string*⟩ that we want to output. The ⟨*other string*⟩ is built from the ⟨*token list*⟩ by `f`-expanding `\__str_to_other:n`.

```
\cs_new:Npn \__str_format_tl_s:NNnnNNn #1#2#3#4#5#6#7
 {
  \exp_args:Nc \exp_args:Nf
   { __str_format_align_#6:nnnN }
   { \exp_args:Nf #2 { \__str_to_other:n {#7} } #3 }
   { }
   {#4} #5
 }
```

(*End definition for* `\__str_format_tl_s:NNnnNNn`.)

4.5 Formatting sequences

`\seq_format:Nn`
`\seq_format:cn`

Each item is formatted as a token list according to the specification. First parse the format and expand the sequence, then loop through the items. Eventually, convert to a string.

```
\cs_new:Npn \seq_format:Nn #1#2
 {
  \tl_to_str:f
   { \__str_format_seq:of {#1} { \__str_format_parse:n {#2} } }
 }
\cs_generate_variant:Nn \seq_format:Nn { c }
```

(*End definition for* `\seq_format:Nn` *and* `\seq_format:cn`*. These functions are documented on page* **??**.)

`\__str_format_seq:nn`
`\__str_format_seq:of`

The first argument is the contents of a `seq` variable. The second is a parsed ⟨*format specification*⟩. Set up the loop.

```
\cs_new:Npn \__str_format_seq:nn #1#2
 {
  \__str_format_seq_loop:nnNn { } {#2}
   #1
   { ? \__str_format_seq_end:w } { }
 }
\cs_generate_variant:Nn \__str_format_seq:nn { of }
```

(*End definition for* `\__str_format_seq:nn` *and* `\__str_format_seq:of`.)

__str_format_seq_loop:nnNn

`\__str_format_seq_loop:nnNn {⟨done⟩} {⟨parsed format⟩} \__seq_item:n {⟨item⟩}`

The first argument is the result of formatting the items read so far. The third argument is a single token (`\__seq_item:n`), until we reach the end of the sequence, where `\use_none:n #3` ends the loop.

```
\cs_new:Npn \__str_format_seq_loop:nnNn #1#2#3#4
  {
    \use_none:n #3
    \exp_args:Nf \__str_format_seq_loop:nnNn
      { \use:nf {#1} { \__str_format_tl:NNNnnNn #2 {#4} } }
      {#2}
  }
```

(*End definition for* `\__str_format_seq_loop:nnNn`*.*)

__str_format_seq_end:w

Pick the right piece in the loop above.

```
\cs_new:Npn \__str_format_seq_end:w #1#2#3#4 { \use_ii:nnn #3 }
```

(*End definition for* `\__str_format_seq_end:w`*.*)

4.6 Formatting integers

\int_format:nn

Evalute the first argument and feed it to `\__str_format_int:nn`.

```
\cs_new:Npn \int_format:nn #1
  { \exp_args:Nf \__str_format_int:nn { \int_eval:n {#1} } }
```

(*End definition for* `\int_format:nn`*. This function is documented on page 3.*)

__str_format_int:nn

Parse the ⟨*format specification*⟩ and feed it to `\__str_format_int:NNNnnNn`. Then convert the result to a string

```
\cs_new:Npn \__str_format_int:nn #1#2
  {
    \tl_to_str:f
      {
        \exp_last_unbraced:Nf \__str_format_int:NNNnnNn
          { \__str_format_parse:n {#2} }
          {#1}
      }
  }
```

(*End definition for* `\__str_format_int:nn`*.*)

__str_format_int:NNNnnNn

`\__str_format_int:NNNnnNn ⟨fill⟩ ⟨alignment⟩ ⟨sign⟩ {⟨width⟩} {⟨precision⟩} ⟨style⟩ {⟨integer⟩}`

First set the default alignment `?` to `>`. Place the modified information after a trailing `\s__stop` for later retrieval. Then check the ⟨*sign*⟩: if the integer is negative, always put `-`. Otherwise, if the format's ⟨*sign*⟩ is `~`, put a space (with category "other"); if it is `+` put `+`; if it is `-` (default), put nothing, represented as a brace group. The width `#4` will be useful later, store it after `\s__stop`. Afterwards, check that the ⟨*precision*⟩ was absent. Finally, dispatch depending on the ⟨*style*⟩.

```
\cs_new:Npn \__str_format_int:NNNnnNn #1#2#3#4#5#6#7
```

```
 {
  \token_if_eq_charcode:NNTF #2 ?
   { \__str_format_put:nw { #1 > } }
   { \__str_format_put:nw { #1 #2 } }
  \int_compare:nNnTF {#7} < \c_zero
   { \__str_format_put:nw { - } }
   {
    \str_case:nnn {#3}
     {
      { ~ } { \__str_format_put:ow { \c_catcode_other_space_tl } }
      { + } { \__str_format_put:nw { + } }
     }
     { \__str_format_put:nw { { } } }
   }
  \__str_format_put:nw { {#4} }
  \tl_if_empty:nF {#5}
   {
    \__msg_kernel_expandable_error:nnnn
     { str } { invalid-precision-format } {#5} {int}
   }
  \str_case:nnn {#6}
   {
    { ? } { \__str_format_int:NwnnNNn \use:n }
    { d } { \__str_format_int:NwnnNNn \use:n }
    { b } { \__str_format_int:NwnnNNn \int_to_binary:n }
    { o } { \__str_format_int:NwnnNNn \int_to_octal:n }
    { X } { \__str_format_int:NwnnNNn \int_to_hexadecimal:n }
   }
   {
    \__msg_kernel_expandable_error:nnnn
     { str } { invalid-style-format } {#6} { int }
    \__str_format_int:NwnnNNn \use:n
   }
  \s__stop {#7}
 }
```

(*End definition for* `\__str_format_int:NNNnnNn`.)

__str_format_int:NwnnNNn

`\__str_format_int:NwnnNNn` ⟨*function*⟩ `\s__stop {`⟨*width*⟩`} {`⟨*sign*⟩`}` ⟨*fill*⟩ ⟨*alignment*⟩ `{`⟨*integer*⟩`}`

Use the `format_align` function corresponding to the ⟨*alignment*⟩, with the following arguments:

- the string formed by combining the sign `#4` with the result of converting the absolute value of the ⟨*integer*⟩ `#7` according to the conversion function `#1`;
- the ⟨*width*⟩;
- the ⟨*fill*⟩ character.

```
\cs_new:Npn \__str_format_int:NwnnNNn #1#2 \s__stop #3#4#5#6#7
```

```
  {
    \exp_args:Nc \exp_args:Nf
      { __str_format_align_#6:nnnN }
      { #1 { \int_abs:n {#7} } }
      {#4}
      {#3} #5
  }
```

(*End definition for* `\__str_format_int:NwnnNNn`*.*)

4.7 Formatting floating points

`\fp_format:nn` Evalute the first argument to an internal floating point number, and feed it to `\__str_format_fp:nn`.

```
\cs_new:Npn \fp_format:nn #1
  { \exp_args:Nf \__str_format_fp:nn { \__fp_parse:n {#1} } }
```

(*End definition for* `\fp_format:nn`*. This function is documented on page 3.*)

`\__str_format_fp:nn` Parse the ⟨*format specification*⟩ and feed it to `\__str_format_fp:NNNnnNn`. Then convert the result to a string

```
\cs_new:Npn \__str_format_fp:nn #1#2
  {
    \tl_to_str:f
      {
        \exp_last_unbraced:Nf \__str_format_fp:NNNnnNw
          { \__str_format_parse:n {#2} }
          #1
      }
  }
```

(*End definition for* `\__str_format_fp:nn`*.*)

`\__str_format_fp:NNNnnNw`

`\__str_format_fp:NNNnnNw` ⟨*fill*⟩ ⟨*alignment*⟩ ⟨*format sign*⟩ `{`⟨*width*⟩`}` `{`⟨*precision*⟩`}` ⟨*style*⟩ `\s__fp \__fp_chk:w` ⟨*fp type*⟩ ⟨*fp sign*⟩ ⟨*fp body*⟩ `;`

First set the default alignment `?` to `>`. Place the modified information after a trailing `\s__stop` for later retrieval. Then check the ⟨*format sign*⟩ and the ⟨*fp sign*⟩: if the floating point is negative, always put `-`. Otherwise (including `nan`), if the format's ⟨*sign*⟩ is `~`, put a space (with category "other"); if it is `+` put `+`; if it is `-` (default), put nothing, represented as a brace group. The width `#4` will be useful later, store it after `\s__stop`. Afterwards, check the ⟨*precision*⟩: if it was not given, replace it by 6 (default precision). Finally, dispatch depending on the ⟨*style*⟩.

```
\cs_new:Npn \__str_format_fp:NNNnnNw
    #1#2#3#4#5#6 \s__fp \__fp_chk:w #7 #8
  {
    \token_if_eq_charcode:NNTF #2 ?
      { \__str_format_put:nw { #1 > } }
      { \__str_format_put:nw { #1 #2 } }
    \token_if_eq_meaning:NNTF 2 #8
      { \__str_format_put:nw { - } }
      {
        \str_case:nnn {#3}
          {
            { ~ } { \__str_format_put:ow { \c_catcode_other_space_tl } }
            { + } { \__str_format_put:nw { + } }
          }
          { \__str_format_put:nw { { } } }
      }
    \__str_format_put:nw { {#4} }
    \tl_if_empty:nTF {#5}
      { \__str_format_put:nw { { 6} } }
      { \__str_format_put:nw { {#5} } }
    \str_case:nnn {#6}
      {
        { e } { \__str_format_fp:wnnnNNw \__str_format_fp_e:wn }
        { f } { \__str_format_fp:wnnnNNw \__str_format_fp_f:wn }
        { g } { \__str_format_fp:wnnnNNw \__str_format_fp_g:wn }
        { ? } { \__str_format_fp:wnnnNNw \__str_format_fp_g:wn }
      }
      {
        \__msg_kernel_expandable_error:nnnn
          { str } { invalid-style-format } {#6} { fp }
        \__str_format_fp:wnnnNNw \__str_format_fp_g:wn
      }
    \s__stop
    \s__fp \__fp_chk:w #7 #8
  }
```

(*End definition for* `\__str_format_fp:NNNnnNw`*.*)

__str_format_fp:wnnnNNw

__str_format_fp:wnnnNNw ⟨*formatting function*⟩ \s__stop {⟨*precision*⟩} {⟨*width*⟩} {⟨*sign*⟩} ⟨*fill*⟩ ⟨*alignment*⟩ \s__fp __fp_chk:w ⟨*fp type*⟩ ⟨*fp sign*⟩ ⟨*fp body*⟩ ;

```
\cs_new:Npn \__str_format_fp:wnnnNNw
    #1 \s__stop #2 #3 #4 #5#6 #7 ;
  {
    \exp_args:Nc \exp_args:Nf
      { __str_format_align_#6:nnnN }
      { #1 #7 ; {#2} }
      {#4}
      {#3} #5
  }
```

(*End definition for* `\__str_format_fp:wnnnNNw`*.*)

__str_format_fp_round:wn

Round the given floating point (not its absolute value, to play nicely with unusual rounding modes).

```
\cs_new:Npn \__str_format_fp_round:wn #1 ; #2
  { \__fp_parse:n { round ( #1; , #2 - \__fp_exponent:w #1; ) } }
```

(*End definition for* `\__str_format_fp_round:wn`*.*)

__str_format_fp_e:wn
__str_format_fp_e_aux:wn

With the e type, first filter out special cases. In the normal case, round to #4+1 significant figures (one before the decimal separator, #4 after).

```
\group_begin:
\char_set_catcode_other:N E
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new:Npn \__str_format_fp_e:wn \s__fp \__fp_chk:w #1#2#3 ; #4
      {
        \int_case:nnn {#1}
          {
            {0} { \use:nf { 0 . } { \prg_replicate:nn {#4} { 0 } } e 0 }
            {2} { inf }
            {3} { nan }
          }
          {
            \exp_last_unbraced:Nf \__str_format_fp_e_aux:wn
              \__str_format_fp_round:wn \s__fp \__fp_chk:w #1#2#3 ; { #4 + 1 }
              {#4}
          }
      }
    \cs_new:Npn \__str_format_fp_e_aux:wn
        \s__fp \__fp_chk:w #1#2 #3 #4#5#6#7 ; #8
      {
        \__str_format_put:fw { \int_eval:n { #3 - 1 } }
        \__str_format_put:nw { e }
        \int_compare:nNnTF {#8} > \c_sixteen
          {
            \__str_format_put:fw { \prg_replicate:nn { #8 - \c_fifteen } {0} }
            \__str_format_put:fw { \use_none:n #4#5#6#7 }
          }
          {
            \__str_format_put:fw
              { \str_range:nnn { #4#5#6#7 0 } { 2 } { #8 + 1 } }
          }
        \__str_format_put:fw { \use_i:nnnn #4 . }
        \use_none:n \s__stop
      }
  }
```

(*End definition for* `\__str_format_fp_e:wn`*. This function is documented on page 3.*)

__str_format_fp_f:wn
__str_format_fp_f_aux:wwwn

With the f type, first filter out special cases. In the normal case, round to #4 (absolute) decimal places.

```
\cs_new:Npn \__str_format_fp_f:wn \s__fp \__fp_chk:w #1#2#3 ; #4
  {
    \int_case:nnn {#1}
      {
        {0} { \use:nf { 0 . } { \prg_replicate:nn {#4} { 0 } } }
        {2} { inf }
        {3} { nan }
      }
      {
        \exp_last_unbraced:Nf \__str_format_fp_f_aux:wwwn
          \fp_to_decimal:n
            { abs ( round ( \s__fp \__fp_chk:w #1#2#3 ; , #4 ) ) }
          . . ;
          {#4}
      }
  }
\cs_new:Npn \__str_format_fp_f_aux:wwwn #1 . #2 . #3 ; #4
  {
    \use:nf
      { #1 . #2 }
      { \prg_replicate:nn { #4 - \__str_count_unsafe:n {#2} } {0} }
  }
```

(*End definition for* `\__str_format_fp_f:wn`*. This function is documented on page 3.*)

`\__str_format_fp_g:wn`
`\__str_format_fp_g_aux:wn`

With the `g` type, first filter out special cases. In the normal case, round to `#4` significant figures, then test the exponent: if $-4 \le \langle \textit{exponent} \rangle < \langle \textit{precision} \rangle$, use the presentation type `f`, otherwise use the presentation type `e`. Also, a $\langle \textit{precision} \rangle$ of 0 is treated like a precision of 1. Actually, we don't reuse the `e` and `f` auxiliaries, because we want to trim trailing zeros. Thankfully, this is done by `\fp_to_decimal:n` and `\fp_to_scientific:n`, acting on the (absolute value of the) rounded value.

```
\cs_new:Npn \__str_format_fp_g:wn \s__fp \__fp_chk:w #1#2 ; #3
  {
    \int_case:nnn {#1}
      {
        {0} { 0 }
        {2} { inf }
        {3} { nan }
      }
      {
        \exp_last_unbraced:Nf \__str_format_fp_g_aux:wn
          \__str_format_fp_round:wn \s__fp \__fp_chk:w #1#2 ;
            { \int_max:nn {1} {#3} }
          { \int_max:nn {1} {#3} }
      }
  }
\cs_new:Npn \__str_format_fp_g_aux:wn #1; #2
  {
    \int_compare:nNnTF { \__fp_exponent:w #1; } < { -3 }
      { \fp_to_scientific:n }
      {
        \int_compare:nNnTF { \__fp_exponent:w #1; } > {#2}
          { \fp_to_scientific:n }
          { \fp_to_decimal:n }
      }
    { \__fp_abs_o:w #1; \prg_do_nothing: }
  }
```

(*End definition for* `\__str_format_fp_g:wn`*. This function is documented on page 3.*)

4.8 Messages

All of the messages are produced expandably, so there is no need for an extra-text.

```
\__msg_kernel_new:nnn { str } { invalid-format }
  { Invalid~format~'#1'. }
\__msg_kernel_new:nnn { str } { invalid-align-format }
  { Invalid~alignment~'#1'~for~type~'#2'. }
\__msg_kernel_new:nnn { str } { invalid-sign-format }
  { Invalid~sign~'#1'~for~type~'#2'. }
\__msg_kernel_new:nnn { str } { invalid-precision-format }
  { Invalid~precision~'#1'~for~type~'#2'. }
\__msg_kernel_new:nnn { str } { invalid-style-format }
  { Invalid~style~'#1'~for~type~'#2'. }
```

4.9 Todos

- Check what happens during floating point formatting when a number is rounded to 0 or ∞. I think the `e` and `f` types break horribly.

```
⟨/initex | package⟩
```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.