Package ‘minty’

January 7, 2025

Title Minimal Type Guesser

Version 0.0.5

Description Port the type guesser from 'readr' (so-called 'readr' first edition parsing engine, now superseded by 'vroom').

License MIT + file LICENSE

URL https://gesistsa.github.io/minty/, https://github.com/gesistsa/minty

BugReports https://github.com/gesistsa/minty/issues

Depends R (>= 3.6)

LinkingTo cpp11 (>= 0.5.0), tzdb (>= 0.1.1)

Config/testthat/edition 3

Config/testthat/parallel false

Encoding UTF-8

Language en-US

RoxygenNote 7.3.2

Imports tzdb

Suggests knitr, stringi, testthat, withr, hms, readr

Config/Needs/website gesistsa/tsatemplate

NeedsCompilation yes

Author Chung-hong Chan [aut, cre] (<https://orcid.org/0000-0002-6232-7530>),
Hadley Wickham [aut] (author of the ported code from readr),
Jim Hester [aut] (author of the ported code from readr),
Romain Francois [ctb] (author of the ported code from readr),
Jennifer Bryan [aut] (author of the ported code from readr),
Shelby Bearrows [ctb] (author of the ported code from readr),
Posit Software, PBC [cph] (copyright holder of readr),
David Olson [aut] (author of src/tzfile.h)

Maintainer Chung-hong Chan <chainsawtiney@gmail.com>

Repository CRAN

Date/Publication 2025-01-07 14:30:07 UTC

Contents

col_skip *Create column specification*

Description

`cols()` includes all columns in the input data, guessing the column types as the default. `cols_only()` includes only the columns you explicitly specify, skipping the rest. In general you can substitute `list()` for `cols()` without changing the behavior.

Usage

```
col_skip()

cols(..., .default = col_guess())

cols_only(...)
```

Arguments

`...`	Either column objects created by `col_*()`, or their abbreviated character names (as described in the `col_types` argument of read_delim). If you're only overriding a few columns, it's best to refer to columns by name. If not named, the column types must match the column names exactly.
`.default`	Any named columns not explicitly overridden in `...` will be read with this column type.

Details

The available specifications are: (with string abbreviations in brackets)

- `col_logical()` [l], containing only `T`, `F`, `TRUE` or `FALSE`.
- `col_integer()` [i], integers.
- `col_double()` [d], doubles.
- `col_character()` [c], everything else.

- `col_factor(levels, ordered)` [f], a fixed set of values.
- `col_date(format = "")` [D]: with the locale's `date_format`.
- `col_time(format = "")` [t]: with the locale's `time_format`.
- `col_datetime(format = "")` [T]: ISO8601 date times
- `col_number()` [n], numbers containing the `grouping_mark`
- `col_skip()` [_, -], don't import this column.
- `col_guess()` [?], parse using the "best" type based on the input.

Value

a list / S3 object representing column specification

See Also

Other parsers: `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_number()`, `parse_vector()`

Other parsers: `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_number()`, `parse_vector()`

Examples

```
cols(a = col_integer())
cols_only(a = col_integer())

# You can also use the standard abbreviations
cols(a = "i")
cols(a = "i", b = "d", c = "_")

# You can also use multiple sets of column definitions by combining
# them like so:

t1 <- cols(
  column_one = col_integer(),
  column_two = col_number()
)

t2 <- cols(
  column_three = col_character()
)

t3 <- t1
t3$cols <- c(t1$cols, t2$cols)
t3
```

date_names — *Create or retrieve date names*

Description

When parsing dates, you often need to know how weekdays of the week and months are represented as text. This pair of functions allows you to either create your own, or retrieve from a standard list. The standard list is derived from ICU (`http://site.icu-project.org`) via the stringi package.

Usage

```
date_names(mon, mon_ab = mon, day, day_ab = day, am_pm = c("AM", "PM"))

date_names_lang(language)

date_names_langs()
```

Arguments

`mon`, `mon_ab`	Full and abbreviated month names.
`day`, `day_ab`	Full and abbreviated week day names. Starts with Sunday.
`am_pm`	Names used for AM and PM.
`language`	A BCP 47 locale, made up of a language and a region, e.g. `"en"` for American English. See `date_names_langs()` for a complete list of available locales.

Value

a list / S3 object representing data time

Examples

```
date_names_lang("en")
date_names_lang("ko")
date_names_lang("fr")
```

locale — *Create locales*

Description

A locale object tries to capture all the defaults that can vary between countries. You set the locale in once, and the details are automatically passed on down to the columns parsers. The defaults have been chosen to match R (i.e. US English) as closely as possible.

Usage

```
locale(
  date_names = "en",
  date_format = "%AD",
  time_format = "%AT",
  decimal_mark = ".",
  grouping_mark = ",",
  tz = "UTC",
  encoding = "UTF-8",
  asciify = FALSE
)

default_locale()
```

Arguments

`date_names`	Character representations of day and month names. Either the language code as string (passed on to `date_names_lang()`) or an object created by `date_names()`.
`date_format`, `time_format`	Default date and time formats.
`decimal_mark`, `grouping_mark`	Symbols used to indicate the decimal place, and to chunk larger numbers. Decimal mark can only be `,` or `.`.
`tz`	Default tz. This is used both for input (if the time zone isn't present in individual strings), and for output (to control the default display). The default is to use "UTC", a time zone that does not use daylight savings time (DST) and hence is typically most useful for data. The absence of time zones makes it approximately 50x faster to generate UTC times than any other time zone. Use `""` to use the system default time zone, but beware that this will not be reproducible across systems. For a complete list of possible time zones, see `OlsonNames()`. Americans, note that "EST" is a Canadian time zone that does not have DST. It is *not* Eastern Standard Time. It's better to use "US/Eastern", "US/Central" etc.
`encoding`	Default encoding (not used in `minty`).
`asciify`	Should diacritics be stripped from date names and converted to ASCII? This is useful if you're dealing with ASCII data where the correct spellings have been lost. Requires the **stringi** package.

Value

a list / S3 object representing the locale information

Examples

```
locale()
locale("fr")
```

```
# South American locale
locale("es", decimal_mark = ",")
```

---

parse_atomic *Parse logicals, integers, and reals*

Description

Use `parse_*()` if you have a character vector you want to parse.

Usage

```
parse_logical(
  x,
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE,
  .return_problems = FALSE
)

parse_integer(
  x,
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE,
  .return_problems = FALSE
)

parse_double(
  x,
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE,
  .return_problems = FALSE
)

parse_character(
  x,
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE,
  .return_problems = FALSE
)

col_logical()

col_integer()
```

```
col_double()

col_character()
```

Arguments

`x`	Character vector of values to parse.
`na`	Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
`locale`	The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
`trim_ws`	Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it?
`.return_problems`	Whether to hide the `problems` tibble from the output

Value

a parsed vector

See Also

Other parsers: `col_skip()`, `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_number()`, `parse_vector()`

Examples

```
parse_integer(c("1", "2", "3"))
parse_double(c("1", "2", "3.123"))
parse_number("$1,123,456.00")

# Use locale to override default decimal and grouping marks
es_MX <- locale("es", decimal_mark = ",")
parse_number("$1.123.456,00", locale = es_MX)

# Invalid values are replaced with missing values with a warning.
x <- c("1", "2", "3", "-")
parse_double(x)
# Or flag values as missing
parse_double(x, na = "-")
```

`parse_datetime` *Parse date/times*

Description

Parse date/times

Usage

```
parse_datetime(
  x,
  format = "",
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE,
  .return_problems = FALSE
)

parse_date(
  x,
  format = "",
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE,
  .return_problems = FALSE
)

parse_time(
  x,
  format = "",
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE,
  .return_problems = FALSE
)

col_datetime(format = "")

col_date(format = "")

col_time(format = "")
```

Arguments

x	A character vector of dates to parse.

`format`	A format specification, as described below. If set to "", date times are parsed as ISO8601, dates and times used the date and time formats specified in the `locale()`. Unlike `strptime()`, the format specification must match the complete string.
`na`	Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
`locale`	The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
`trim_ws`	Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it?
`.return_problems`	Whether to hide the `problems` tibble from the output

Value

A `POSIXct()` vector with `tzone` attribute set to `tz`. Elements that could not be parsed (or did not generate valid dates) will be set to `NA`, and a warning message will inform you of the total number of failures.

Format specification

`minty` (inherited from `readr`) uses a format specification similar to `strptime()`. There are three types of element:

1. Date components are specified with "%" followed by a letter. For example "%Y" matches a 4 digit year, "%m", matches a 2 digit month and "%d" matches a 2 digit day. Month and day default to `1`, (i.e. Jan 1st) if not present, for example if only a year is given.
2. Whitespace is any sequence of zero or more whitespace characters.
3. Any other character is matched exactly.

`parse_datetime()` recognises the following format specifications:

- Year: "%Y" (4 digits). "%y" (2 digits); 00-69 -> 2000-2069, 70-99 -> 1970-1999.
- Month: "%m" (2 digits), "%b" (abbreviated name in current locale), "%B" (full name in current locale).
- Day: "%d" (2 digits), "%e" (optional leading space), "%a" (abbreviated name in current locale).
- Hour: "%H" or "%I" or "%h", use I (and not H) with AM/PM, use h (and not H) if your times represent durations longer than one day.
- Minutes: "%M"
- Seconds: "%S" (integer seconds), "%OS" (partial seconds)
- Time zone: "%Z" (as name, e.g. "America/Chicago"), "%z" (as offset from UTC, e.g. "+0800")
- AM/PM indicator: "%p".

- Non-digits: "%." skips one non-digit character, "%+" skips one or more non-digit characters, "%*" skips any number of non-digits characters.
- Automatic parsers: "%AD" parses with a flexible YMD parser, "%AT" parses with a flexible HMS parser.
- Time since the Unix epoch: "%s" decimal seconds since the Unix epoch.
- Shortcuts: "%D" = "%m/%d/%y", "%F" = "%Y-%m-%d", "%R" = "%H:%M", "%T" = "%H:%M:%S", "%x" = "%y/%m/%d".

ISO8601 support

Currently, `minty` does not support all of ISO8601. Missing features:

- Week & weekday specifications, e.g. "2013-W05", "2013-W05-10".
- Ordinal dates, e.g. "2013-095".
- Using commas instead of a period for decimal separator.

The parser is also a little laxer than ISO8601:

- Dates and times can be separated with a space, not just T.
- Mostly correct specifications like "2009-05-19 14:" and "200912-01" work.

See Also

Other parsers: `col_skip()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_number()`, `parse_vector()`

Examples

```
# Format strings --------------------------------------------------------
parse_datetime("01/02/2010", "%d/%m/%Y")
parse_datetime("01/02/2010", "%m/%d/%Y")
# Handle any separator
parse_datetime("01/02/2010", "%m%.%d%.%Y")

# Dates look the same, but internally they use the number of days since
# 1970-01-01 instead of the number of seconds. This avoids a whole lot
# of troubles related to time zones, so use if you can.
parse_date("01/02/2010", "%d/%m/%Y")
parse_date("01/02/2010", "%m/%d/%Y")

# You can parse timezones from strings (as listed in OlsonNames())
parse_datetime("2010/01/01 12:00 US/Central", "%Y/%m/%d %H:%M %Z")
# Or from offsets
parse_datetime("2010/01/01 12:00 -0600", "%Y/%m/%d %H:%M %z")

# Use the locale parameter to control the default time zone
# (but note UTC is considerably faster than other options)
parse_datetime("2010/01/01 12:00", "%Y/%m/%d %H:%M",
  locale = locale(tz = "US/Central")
)
```

```
parse_datetime("2010/01/01 12:00", "%Y/%m/%d %H:%M",
  locale = locale(tz = "US/Eastern")
)

# Unlike strptime, the format specification must match the complete
# string (ignoring leading and trailing whitespace). This avoids common
# errors:
strptime("01/02/2010", "%d/%m/%y")
parse_datetime("01/02/2010", "%d/%m/%y")

# Failures -------------------------------------------------------------
parse_datetime("01/01/2010", "%d/%m/%Y")
parse_datetime(c("01/ab/2010", "32/01/2010"), "%d/%m/%Y")

# Locales --------------------------------------------------------------
# By default, readr expects English date/times, but that's easy to change'
parse_datetime("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
parse_datetime("1 enero 2015", "%d %B %Y", locale = locale("es"))

# ISO8601 --------------------------------------------------------------
# With separators
parse_datetime("1979-10-14")
parse_datetime("1979-10-14T10")
parse_datetime("1979-10-14T10:11")
parse_datetime("1979-10-14T10:11:12")
parse_datetime("1979-10-14T10:11:12.12345")

# Without separators
parse_datetime("19791014")
parse_datetime("19791014T101112")

# Time zones
us_central <- locale(tz = "US/Central")
parse_datetime("1979-10-14T1010", locale = us_central)
parse_datetime("1979-10-14T1010-0500", locale = us_central)
parse_datetime("1979-10-14T1010Z", locale = us_central)
# Your current time zone
parse_datetime("1979-10-14T1010", locale = locale(tz = ""))
```

---

parse_factor *Parse factors*

Description

`parse_factor()` is similar to `factor()`.

Usage

```
parse_factor(
  x,
  levels = NULL,
  ordered = FALSE,
  na = c("", "NA"),
  locale = default_locale(),
  include_na = TRUE,
  trim_ws = TRUE,
  .return_problems = FALSE
)

col_factor(levels = NULL, ordered = FALSE, include_na = FALSE)
```

Arguments

`x`	Character vector of values to parse.
`levels`	Character vector of the allowed levels. When `levels = NULL` (the default), `levels` are discovered from the unique values of `x`, in the order in which they appear in `x`.
`ordered`	Is it an ordered factor?
`na`	Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
`locale`	The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
`include_na`	If `TRUE` and `x` contains at least one `NA`, then `NA` is included in the levels of the constructed factor.
`trim_ws`	Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it?
`.return_problems`	Whether to hide the `problems` tibble from the output

Value

a parsed vector

See Also

Other parsers: `col_skip()`, `parse_datetime()`, `parse_guess()`, `parse_logical()`, `parse_number()`, `parse_vector()`

Examples

```
# discover the levels from the data
parse_factor(c("a", "b"))
parse_factor(c("a", "b", "-99"))
parse_factor(c("a", "b", "-99"), na = c("", "NA", "-99"))
parse_factor(c("a", "b", "-99"), na = c("", "NA", "-99"), include_na = FALSE)
```

```
# provide the levels explicitly
parse_factor(c("a", "b"), levels = letters[1:5])

x <- c("cat", "dog", "caw")
animals <- c("cat", "dog", "cow")

# base::factor() silently converts elements that do not match any levels to
# NA
factor(x, levels = animals)

# parse_factor() generates same factor as base::factor() but throws a warning
# and reports problems
parse_factor(x, levels = animals)
```

---

parse_guess *Parse using the "best" type*

Description

parse_guess() returns the parser vector. This function uses a number of heuristics to determine which type of vector is "best". Generally they try to err of the side of safety, as it's straightforward to override the parsing choice if needed.

Usage

```
parse_guess(
  x,
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE,
  guess_integer = FALSE,
  guess_max = NA,
  .return_problems = FALSE
)

col_guess()
```

Arguments

x	Character vector of values to parse.
na	Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values.
locale	The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
trim_ws	Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it?

`guess_integer`	If `TRUE`, guess integer types for whole numbers, if `FALSE` guess numeric type for all numbers.
`guess_max`	Maximum number of data rows to use for guessing column types. `NA`: uses all data.
`.return_problems`	Whether to hide the `problems` tibble from the output

Value

a parsed vector

See Also

Other parsers: `col_skip()`, `parse_datetime()`, `parse_factor()`, `parse_logical()`, `parse_number()`, `parse_vector()`

Examples

```
# Logical vectors
parse_guess(c("FALSE", "TRUE", "F", "T"))

# Integers and doubles
parse_guess(c("1", "2", "3"))
parse_guess(c("1.6", "2.6", "3.4"))

# Numbers containing grouping mark
parse_guess("1,234,566")

# ISO 8601 date times
parse_guess(c("2010-10-10"))
```

`parse_number` *Parse numbers, flexibly*

Description

This parses the first number it finds, dropping any non-numeric characters before the first number and all characters after the first number. The grouping mark specified by the locale is ignored inside the number.

Usage

```
parse_number(
  x,
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE,
  .return_problems = FALSE
)

col_number()
```

Arguments

`x`	Character vector of values to parse.
`na`	Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
`locale`	The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
`trim_ws`	Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it?
`.return_problems`	Whether to hide the `problems` tibble from the output

Value

A numeric vector (double) of parsed numbers.

a parsed vector

See Also

Other parsers: `col_skip()`, `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_vector()`

Examples

```
## These all return 1000
parse_number("$1,000") ## leading `$` and grouping character `,` ignored
parse_number("euro1,000") ## leading non-numeric euro ignored
parse_number("t1000t1000") ## only parses first number found

parse_number("1,234.56")
## explicit locale specifying European grouping and decimal marks
parse_number("1.234,56", locale = locale(decimal_mark = ",", grouping_mark = "."))
## SI/ISO 31-0 standard spaces for number grouping
parse_number("1 234.56", locale = locale(decimal_mark = ".", grouping_mark = " "))

## Specifying strings for NAs
parse_number(c("1", "2", "3", "NA"))
parse_number(c("1", "2", "3", "NA", "Nothing"), na = c("NA", "Nothing"))
```

`type_convert` *Re-convert character columns in existing data frame*

Description

This is useful if you need to do some manual munging - you can read the columns in as character, clean it up with (e.g.) regular expressions and then let readr take another stab at parsing it. The name is a homage to the base `utils::type.convert()`.

Usage

```
type_convert(
  df,
  col_types = NULL,
  na = c("", "NA"),
  trim_ws = TRUE,
  locale = default_locale(),
  guess_integer = FALSE,
  guess_max = NA,
  verbose = FALSE
)
```

Arguments

`df`	A data frame.
`col_types`	One of `NULL`, a `cols()` specification, or a string. If `NULL`, column types will be imputed using all rows.
`na`	Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
`trim_ws`	Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it?
`locale`	The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
`guess_integer`	If `TRUE`, guess integer types for whole numbers, if `FALSE` guess numeric type for all numbers.
`guess_max`	Maximum number of data rows to use for guessing column types. `NA`: uses all data.
`verbose`	whether to print messages

Value

A data frame

Note

`type_convert()` removes a 'spec' attribute (if it presents).

Examples

```
df <- data.frame(
  x = as.character(runif(10)),
  y = as.character(sample(10)),
  stringsAsFactors = FALSE
)
str(df)
str(type_convert(df))

df <- data.frame(x = c("NA", "10"), stringsAsFactors = FALSE)
str(type_convert(df))
```

Index