Package: tabr (via r-universe)

June 30, 2024

Title Music Notation Syntax, Manipulation, Analysis and Transcription in R

Version 0.5.0

Description Provides a music notation syntax and a collection of music programming functions for generating, manipulating, organizing, and analyzing musical information in R. Music syntax can be entered directly in character strings, for example to quickly transcribe short pieces of music. The package contains functions for directly performing various mathematical, logical and organizational operations and musical transformations on special object classes that facilitate working with music data and notation. The same music data can be organized in tidy data frames for a familiar and powerful approach to the analysis of large amounts of structured music data. Functions are available for mapping seamlessly between these formats and their representations of musical information. The package also provides an API to 'LilyPond' (<https://lilypond.org/>) for transcribing musical representations in R into tablature (``tabs") and sheet music. 'LilyPond' is open source music engraving software for generating high quality sheet music based on markup syntax. The package generates 'LilyPond' files from R code and can pass them to the 'LilyPond' command line interface to be rendered into sheet music PDF files or inserted into R markdown documents. The package offers nominal MIDI file output support in conjunction with rendering sheet music. The package can read MIDI files and attempts to structure the MIDI data to integrate as best as possible with the data structures and functionality found throughout the package.

License MIT + file LICENSE

URL https://github.com/leonawicz/tabr

BugReports https://github.com/leonawicz/tabr/issues

Depends R (>= 2.10)

Imports crayon, dplyr, ggplot2, purrr, tibble, tidyr

Suggests fansi, gridExtra, htmltools, kableExtra, knitr, png, rmarkdown, testthat, tuneR

VignetteBuilder knitr

ByteCompile true

Encoding UTF-8

LazyData true

RoxygenNote 7.2.3

SystemRequirements LilyPond v2.22.1-2+ (needed for rendering sheet music or writing MIDI files)

NeedsCompilation no

Author Matthew Leonawicz [aut, cre] (<https://orcid.org/0000-0001-9452-2771>)

Maintainer Matthew Leonawicz <mfleonawicz@gmail.com>

Repository CRAN

Date/Publication 2024-06-29 17:20:02 UTC

Contents

`append_phrases` *Concatenate and repeat*

Description

Helper functions for concatenating musical phrases and raw strings together as well as repetition.

Usage

```
pc(...)

pn(x, n = 1)
```

Arguments

`...`	character, phrase or non-phrase string.
`x`	character, phrase or non-phrase string.
`n`	integer, number of repetitions.

Details

Note: When working with special `tabr` classes, you can simply use generics like `c()` and `rep()` as many custom methods exist for these classes. The additional respective helper functions, `pc()` and `pn()`, are more specifically for phrase objects and when you are still working with character strings, yet to be converted to a phrase object (numbers not yet in string form are allowed). See examples.

The functions `pc()` and `pn()` are based on base functions `paste()` and `rep()`, respectively, but are tailored for efficiency in creating musical phrases.

These functions respect and retain the phrase class when applied to phrases. They are aggressive for phrases and secondarily for noteworthy strings. Combining a phrase with a non-phrase string will assume compatibility and result in a new phrase object. If no phrase objects are present, the presence of any noteworthy string will in turn attempt to force conversion of all strings to noteworthy strings. The aggressiveness provides convenience, but is counter to expected coercion rules. It is up to the user to ensure all inputs can be forced into the more specific child class.

This is especially useful for repeated instances. This function applies to general slur notation as well. Multiple input formats are allowed. Total number of note durations must be even because all slurs require start and stop points.

Value

phrase on non-phrase character string, noteworthy string if applicable.

Examples

```
pc(8, "16-", "8^")
pn(1, 2)
x <- phrase("c ec'g' ec'g'", "4 4 2", "5 432 432")
y <- phrase("a", 1, 5)
pc(x, y)
pc(x, pn(y, 2))
pc(x, "r1") # add a simple rest instance
class(pc(x, y))
class(pn(y, 2))
class(pc(x, "r1"))
class(pn("r1", 2))
class(pc("r1", "r4"))
```

---

articulations *Single note articulations and syntax*

Description

A data frame containing categorized sets of articulations that can be used in phrase construction.

Usage

```
articulations
```

Format

A data frame with 3 column and 44 rows.

---

as_music_df *Noteworthy string to data frame*

Description

Convert a noteworthy string to a tibble data frame and include additional derivative variables.

Usage

```
as_music_df(
  notes,
  info = NULL,
  key = NULL,
  scale = "diatonic",
  chords = c("root", "list", "character"),
  si_format = c("mmp_abb", "mmp", "ad_abb", "ad")
)
```

Arguments

notes	character, a noteworthy string. Alternatively, a music object or a phrase object, in which case info is ignored.
info	NULL or character, a note info string.
key	character, key signature, only required for inclusion of scale degrees.
scale	character, defaults to "diatonic". Only used in conjunction with key, this can be used to alter scale degrees. Not any arbitrary combination of valid key and valid scale is valid. See scale_degree().
chords	character, how to structure columns containing multiple values per chord/row of data frame. See details.
si_format	character, format for scale intervals. See scale_interval().

Details

If info is provided or notes is a phrase object, the resulting data frame also contains note durations and other info variables. The duration column is always included in the output even as a vector of NAs when info = NULL. This makes it more explicit that a given music data frame was generated without any time information for the timesteps. Other note info columns are not included in this case.

For some derived column variables the root note (lowest pitch) in chord is used. This is done for pitch intervals and scale intervals between adjacent timesteps. This also occurs for scale degrees.

chord = "root" additionally collapses columns like semitone, octave, and frequency to the value for the root note so that all rows contain one numeric value. chord = "list" retains full information as list columns. chord = "character" collapses into strings so that values are readily visible when printing the table, but information is not stripped and can be recovered without recomputing from the original pitches.

Value

a tibble data frame

Examples

```
x <- "a, b, c d e f g# a r ac'e' a c' e' c' r r r a"
as_music_df(x, key = "c", scale = "major")
as_music_df(x, key = "am", scale = "harmonic_minor", si_format = "ad_abb")
```

```
a <- notate("8", "Start here.")
time <- paste(a, "8^*2 16-_ 4.. 16( 16)( 2) 2 4. t8- t8 t8- 8[accent]*4 1")
d1 <- as_music_df(x, time)
d1

# Go directly from music object to data frame
m1 <- as_music(x, time)
d2 <- as_music_df(m1)
identical(d1, d2)

# Go directly from phrase object to data frame
p1 <- phrase("a b cgc'", "4-+ 4[accent] 2", 5)
identical(as_music_df(as_music("a4-+;5 b[accent] cgc'2")), as_music_df(p1))
```

---

chord-compare *Rank, order and sort chords and notes*

Description

Rank, order and sort chords and notes by various definitions.

Usage

```
chord_rank(notes, pitch = c("min", "mean", "max"), ...)

chord_order(notes, pitch = c("min", "mean", "max"), ...)

chord_sort(notes, pitch = c("min", "mean", "max"), decreasing = FALSE, ...)
```

Arguments

notes	character, a noteworthy string.
pitch	character, how ranking of chords is determined; lowest pitch, mean pitch, or highest pitch.
...	additional arguments passed to rank() or order().
decreasing	logical, sort in decreasing order.

Details

There are three options for comparing the relative pitch position of chords provided: comparison of the lowest or root note of each chord, the highest pitch note, or taking the mean of all notes in a chord.

Value

integer for rank and order, character for sort

Examples

```
x <- "a2 c a2 ceg ce_g cea"
chord_rank(x, "min")
chord_rank(x, "max")
chord_rank(x, "mean")

chord_order(x)
chord_order(x, "mean")
chord_sort(x, "mean")
```

`chord-filter` *Extract notes from chords*

Description

Filter or slice chords to extract individual notes.

Usage

```
chord_root(notes)

chord_top(notes)

chord_slice(notes, index)
```

Arguments

`notes`	character, a noteworthy string.
`index`	integer, the order of a note in a chord by pitch (not scale degrees).

Details

These functions extract notes from chords such as the root note, the highest pitch, specific position among the notes by pitch, or trim chords to simplify them. They operate based only on ordered pitches.

For `chord_slice()`, any entry that is empty after slicing is dropped. An error is thrown is `index` is completely out of bounds for all chords.

Value

a noteworthy string

Examples

```
x <- "a_2 c#eg# e_gc egc,cc'"
chord_root(x)
chord_top(x)
identical(chord_slice(x, 1), chord_root(x))
chord_slice(x, 2)
chord_slice(x, 4)
chord_slice(x, 3:5)
```

---

chord-mapping *Chord mapping*

Description

Helper functions for chord mapping.

Usage

```
gc_info(
  name,
  root_octave = NULL,
  root_fret = NULL,
  min_fret = NULL,
  bass_string = NULL,
  open = NULL,
  key = "c",
  ignore_octave = TRUE
)

gc_fretboard(
  name,
  root_octave = NULL,
  root_fret = NULL,
  min_fret = NULL,
  bass_string = NULL,
  open = NULL,
  key = "c",
  ignore_octave = TRUE
)

gc_notes_to_fb(
  notes,
  root_octave = NULL,
  root_fret = NULL,
  min_fret = NULL,
  bass_string = NULL,
  open = NULL
```

```
)

gc_notes(
  name,
  root_octave = NULL,
  root_fret = NULL,
  min_fret = NULL,
  bass_string = NULL,
  open = NULL,
  key = "c",
  ignore_octave = TRUE
)

gc_is_known(notes)

gc_name_split(name)

gc_name_root(name)

gc_name_mod(name)
```

Arguments

name	character, chord name in tabr format, e.g., "bM b_m b_m7#5", etc.
root_octave	integer, optional filter for chords whose root note is in a set of octave numbers. May be a vector.
root_fret	integer, optional filter for chords whose root note matches a specific fret. May be a vector.
min_fret	integer, optional filter for chords whose notes are all at or above a specific fret. May be a vector.
bass_string	integer, optional filter for chords whose lowest pitch string matches a specific string, 6, 5, or 4. May be a vector.
open	logical, optional filter for open and movable chords. NULL retains both types.
key	character, key signature, used to enforce type of accidentals.
ignore_octave	logical, if TRUE, functions like gc_info() and gc_fretboard() return more results.
notes	character, a noteworthy string.

Details

These functions assist with mapping between different information that define chords.

For gc_is_known(), a check is done against chords in the guitarChords dataset. A simple noteworthy string is permitted, but any single-note entry will automatically yield a FALSE result.

gc_info() returns a tibble data frame containing complete information for the subset of predefined guitar chords specified by name and key. Any accidentals present in the chord root of name (but not in the chord modifier, e.g., m7_5 or m7#5) are converted according to key if necessary. gc_notes()

and gc_fretboard() are wrappers around gc_info(), which return noteworthy strings of chord notes and a named vector of LilyPond fretboard diagram data, respectively. Note that although the input to these functions can contain multiple chord names, whether as a vector or as a single space-delimited string, the result is not intended to be of equal length. These functions filter guitarChords. The result is the set of all chords matched by the supplied input filters.

gc_name_split() splits a vector or space-delimited set of chord names into a tibble data frame containing separate chord root and chord modifier columns. gc_name_root() and gc_name_mod() are wrappers around this.

Value

various, see details regarding each function.

Examples

```
gc_is_known("a b_,fb_d'f'")

gc_name_root("a aM b_,m7#5")
gc_name_mod("a aM b_,m7#5")

gc_info("a") # a major chord, not a single note
gc_info("ceg a#m7_5") # only second entry is a guitar chord
gc_info("ceg a#m7_5", key = "f")

gc_info("a,m c d f,")
gc_fretboard("a,m c d f,", root_fret = 0:3)
gc_notes_to_fb("a,eac'e' cgc'e'g'")

x <- gc_notes("a, b,", root_fret = 0:2)
summary(x)
```

---

chords *Chord constructors*

Description

These functions construct basic chord string notation from root notes.

Usage

```
chord_min(notes, key = "c", octaves = "tick")

chord_maj(notes, key = "c", octaves = "tick")

chord_min7(notes, key = "c", octaves = "tick")

chord_dom7(notes, key = "c", octaves = "tick")
```

```
chord_7s5(notes, key = "c", octaves = "tick")

chord_maj7(notes, key = "c", octaves = "tick")

chord_min6(notes, key = "c", octaves = "tick")

chord_maj6(notes, key = "c", octaves = "tick")

chord_dim(notes, key = "c", octaves = "tick")

chord_dim7(notes, key = "c", octaves = "tick")

chord_m7b5(notes, key = "c", octaves = "tick")

chord_aug(notes, key = "c", octaves = "tick")

chord_5(notes, key = "c", octaves = "tick")

chord_sus2(notes, key = "c", octaves = "tick")

chord_sus4(notes, key = "c", octaves = "tick")

chord_dom9(notes, key = "c", octaves = "tick")

chord_7s9(notes, key = "c", octaves = "tick")

chord_maj9(notes, key = "c", octaves = "tick")

chord_add9(notes, key = "c", octaves = "tick")

chord_min9(notes, key = "c", octaves = "tick")

chord_madd9(notes, key = "c", octaves = "tick")

chord_min11(notes, key = "c", octaves = "tick")

chord_7s11(notes, key = "c", octaves = "tick")

chord_maj7s11(notes, key = "c", octaves = "tick")

chord_11(notes, key = "c", octaves = "tick")

chord_maj11(notes, key = "c", octaves = "tick")

chord_13(notes, key = "c", octaves = "tick")

chord_min13(notes, key = "c", octaves = "tick")
```

```
chord_maj13(notes, key = "c", octaves = "tick")

xm(notes, key = "c", octaves = "tick")

xM(notes, key = "c", octaves = "tick")

xm7(notes, key = "c", octaves = "tick")

x7(notes, key = "c", octaves = "tick")

x7s5(notes, key = "c", octaves = "tick")

xM7(notes, key = "c", octaves = "tick")

xm6(notes, key = "c", octaves = "tick")

xM6(notes, key = "c", octaves = "tick")

xdim(notes, key = "c", octaves = "tick")

xdim7(notes, key = "c", octaves = "tick")

xm7b5(notes, key = "c", octaves = "tick")

xaug(notes, key = "c", octaves = "tick")

x5(notes, key = "c", octaves = "tick")

xs2(notes, key = "c", octaves = "tick")

xs4(notes, key = "c", octaves = "tick")

x9(notes, key = "c", octaves = "tick")

x7s9(notes, key = "c", octaves = "tick")

xM9(notes, key = "c", octaves = "tick")

xadd9(notes, key = "c", octaves = "tick")

xm9(notes, key = "c", octaves = "tick")

xma9(notes, key = "c", octaves = "tick")

xm11(notes, key = "c", octaves = "tick")

x7s11(notes, key = "c", octaves = "tick")
```

```
xM7s11(notes, key = "c", octaves = "tick")

x_11(notes, key = "c", octaves = "tick")

xM11(notes, key = "c", octaves = "tick")

x_13(notes, key = "c", octaves = "tick")

xm13(notes, key = "c", octaves = "tick")

xM13(notes, key = "c", octaves = "tick")
```

Arguments

notes	character, a noteworthy string of chord root notes.
key	key signature. See details.
octaves	character, passed to `transpose()`.

Details

Providing a key signature is used only to ensure flats or sharps for accidentals. An additional set of aliases with efficient names, of the form `x*` where `*` is a chord modifier abbreviation, is provided to complement the set of `chord_*` functions.

These functions create standard chords, not the multi-octave spanning types of chords commonly played on guitar.

Value

character

See Also

`transpose()`

Examples

```
chord_min("d")
chord_maj("d")
xM("d")
xm("c f g")
xm("c, f, g,", key = "e_")
```

chord_arpeggiate *Arpeggiate a chord*

Description

Create an arpeggio from a chord.

Usage

```
chord_arpeggiate(
  chord,
  n = 0,
  by = c("note", "chord"),
  broken = FALSE,
  collapse = FALSE
)
```

Arguments

chord	character, a single chord.
n	integer, number of steps, negative indicates reverse direction (decreasing pitch).
by	whether each of the n steps refers to individual notes in the chord (an inversion) or raising the entire chord in its given position by one octave.
broken	logical, return result as an arpeggio of broken chords.
collapse	logical, collapse result into a single string ready for phrase construction.

Details

This function is based on chord_invert. If n = 0 then chord is returned immediately; other arguments are ignored.

Value

character

Examples

```
chord_arpeggiate("ce_gb_", 2)
chord_arpeggiate("ce_gb_", -2)
chord_arpeggiate("ce_gb_", 2, by = "chord")
chord_arpeggiate("ce_gb_", 1, broken = TRUE, collapse = TRUE)
```

chord_break *Broken chords*

Description

Convert chords in a noteworthy string or vector to broken chords.

Usage

```
chord_break(notes)
```

Arguments

notes	character, noteworthy string that may contain chords.

Value

character

Examples

```
chord_break("c e g ceg ceg")
```

chord_def *Define chords*

Description

Function for creating new chord definition tables.

Usage

```
chord_def(fret, id, optional = NA, tuning = "standard", ...)
```

Arguments

fret	integer vector defining fretted chord. See details.
id	character, the chord type. See details.
optional	NA when all notes required. Otherwise an integer vector giving the indices offret that are considered optional notes for the chord.
tuning	character, string tuning. See tunings for predefined tunings. Custom tunings are specified with a similar value string.
...	additional arguments passed to transpose().

Details

This function creates a tibble data frame containing information defining various attributes of chords. It is used to create the `guitarChords` dataset, but can be used to create other pre-defined chord collections. The tibble has only one row, providing all information for the defined chord. The user can decide which arguments to vectorize over when creating a chord collection. See examples.

This function uses a vector of fret integers (`NA` for muted string) to define a chord, in conjunction with a string `tuning` (defaults to standard tuning, six-string guitar). `fret` is from lowest to highest pitch strings, e.g., strings six through one.

The `id` is passed directly to the output. It represents the type of chord and should conform to accepted `tabr` notation. See `id` column in `guitarChords` for examples.

Note that the `semitones` column gives semitone intervals between chord notes. These count from zero as the lowest pitch based on the tuning of the instrument, e.g., zero is E2 with standard guitar tuning. To convert these semitone intervals to standard semitone values assigned to pitches, use e.g., `pitch_semitones("e2")` (40) if that is the lowest pitch and add that value to the instrument semitone interval values. This is the explanation, but doing this is not necessary. You can use `chord_semitones()` to compute semitones directly on pitches in a chord.

Value

a data frame

Examples

```
frets <- c(NA, 0, 2, 2, 1, 0)
chord_def(frets, "m")
chord_def(frets, "m", 6)

purrr::map_dfr(c(0, 2, 3), ~chord_def(frets + .x, "m"))
```

---

chord_invert *Chord inversion*

Description

This function inverts a single chord given as a character string. If `n = 0`, `chord` is returned immediately. Otherwise, the notes of the chord are inverted. If `abs(n)` is greater than the number of inversions (excluding root position), an error is thrown.

Usage

```
chord_invert(chord, n = 0, limit = FALSE)
```

Arguments

`chord`	character, a single chord.
`n`	inversion.
`limit`	logical, limit inversions in either direction to one less than the number of notes in the chord.

Details

Note that `chord_invert()` has no knowledge of whether a chord might be considered as in root position or some inversion already, as informed by a key signature, chord name or user's intent. This function simply inverts what it receives, treating any defined chord string as in root position.

Octave number applies to this function. Chords should always be defined by notes of increasing pitch. Remember that an unspecified octave number on a note is octave 3. When the chord is inverted, it moves up the scale. The lowest note is moved to the top of the chord, increasing its octave if necessary, to ensure that the note takes the lowest octave number while having the highest pitch. The second lowest note becomes the lowest. It's octave does not change. This pattern is repeated for higher order inversions. The opposite happens if `n` is negative.

The procedure ensures that the resulting inverted chord is still defined by notes of increasing pitch. However, if you construct an unusual chord that spans multiple octaves, the extra space will be condensed by inversion.

Value

character

Examples

```
chord_invert("ce_gb_", 3)
```

---

chord_is_major *Check if chords are major or minor*

Description

Check if chords are major or minor where possible.

Usage

```
chord_is_major(notes)

chord_is_minor(notes)
```

Arguments

`notes`	character, a noteworthy string.

Details

These functions operate based only on ordered pitches. They do not recognize what a human might interpret and name an inverted chord with a root other than the lowest pitch. This imposes limitations on the utility of these functions, which scan the intervals for a minor or major third in a chord whose notes are sorted by pitch.

In several cases including single notes or no major or minor third interval present, `NA` is returned. `TRUE` or `FALSE` is only returned if such an interval is present. If more than one is present, it is based

on the lowest in pitch. It prioritizes major/minor and minor/major adjacent intervals (recognizing a common triad). If these do not occur adjacent, the lowest third is selected. This is still imperfect, but a useful method. Second and higher unknown chord inversions are problematic.

Value

logical vector

Examples

```
x <- "c cg, ce ce_ ceg ce_gb g,ce g,ce_ e_,g,c e_,g,ce_ e_,g,c"
chord_is_major(x)
identical(chord_is_major(x), !chord_is_minor(x))
```

---

chord_set *Generate a chord set*

Description

Generate a chord set for a music score.

Usage

```
chord_set(x, id = NULL, n = 6)
```

Arguments

x	character, n-string chord description from lowest to highest pitch, strings n through 1. E.g., "xo221o". You can use spaces or semicolons when 2-digit fret numbers are present, e.g., "8 10 10 9 o". Do not mix formats. Leading x are inferred if the number of entries is less than n.
id	character, the name of the chord in LilyPond readable format, e.g., "a:m". Ignored if x is already a named vector.
n	number of instrument strings.

Details

The chord set list returned by `chord_set()` is only used for top center placement of a full set of chord fretboard diagrams for a music score. `chord_set()` returns a named list. The names are the chords and the list elements are strings defining string and fret fingering readable by LilyPond. Multiple chord positions can be defined for the same chord name. Instruments with a number of strings other than six are not currently supported.

When defining chords, you may also wish to define rests or silent rests for chords that are to be added to a score for placement above the staff in time, where no chord is to be played or explicitly written. Therefore, there are occasions where you may pass chord names and positions that happen to include entries `r` and/or `s` as `NA` as shown in the example. These two special cases are passed through by `chord_set()` but are ignored when the chord chart is generated.

Value

a named list.

Examples

```
chord_names <- c("e:m", "c", "d", "e:m", "d", "r", "s")
chord_position <- c("997x", "5553x", "7775x", "ooo22o", "232oxx", NA, NA)
chord_set(chord_position, chord_names)
```

`double-bracket` *Double bracket methods for tabr classes*

Description

Double bracket indexing and assignment. See `tabr-methods()` for more details on methods for tabr classes.

Usage

```
## S3 method for class 'noteworthy'
x[[i]]

## S3 method for class 'noteinfo'
x[[i]]

## S3 method for class 'music'
x[[i]]

## S3 method for class 'lyrics'
x[[i]]

## S3 replacement method for class 'noteworthy'
x[[i]] <- value

## S3 replacement method for class 'noteinfo'
x[[i]] <- value

## S3 replacement method for class 'music'
x[[i]] <- value

## S3 replacement method for class 'lyrics'
x[[i]] <- value
```

Arguments

`x`	object.
`i`	index.
`value`	values to assign at index.

See Also

tabr-methods(), note-metadata()

Examples

```
# noteworthy class examples
x <- as_noteworthy("a, b, c ce_g")
x[[3]]
x[[2]] <- paste0(transpose(x[2], 1), "~")
x

# noteinfo class examples
x <- as_noteinfo(c("4-", "t8(", "t8)", "t8x"))
x[[3]]
x[[3]] <- c("t8]")
x

# music class examples
x <- as_music("c,~4 c,1 c'e_'g'4-.*2")
x[[3]]
x[[3]] <- "c'e'g'8"
x
```

---

dyad *Construct a dyad*

Description

Construct a dyad given one note, an interval, and a direction.

Usage

```
dyad(
  notes,
  interval,
  reverse = FALSE,
  octaves = c("tick", "integer"),
  accidentals = c("flat", "sharp"),
  key = NULL
)
```

Arguments

notes	character, a noteworthy string, single notes only, no chords. Number of timesteps must equal the length of interval.
interval	integer or character vector; semitones or interval ID, respectively. See details.
reverse	logical, reverse the transposition direction. Useful when interval is character.
octaves, accidentals, key	See transpose().

Details

The `interval` may be specified by semitones of by common interval name or abbreviation. See examples. For a complete list of valid interval names and abbreviations see `mainIntervals()`. `key` enforces the use of sharps or flats. This function is based on `transpose()`. `notes` and `interval` may be vectors, but must be equal length. Recycling occurs only if one argument is scalar.

Value

character

See Also

`mainIntervals()`

Examples

```
dyad("a", 4)
x <- c("minor third", "m3", "augmented second", "A2")
dyad("a", x)
dyad("c'", x, reverse = TRUE)

x <- c("M3", "m3", "m3", "M3", "M3", "m3", "m3")
dyad(letters[c(3:7, 1, 2)], x)

x <- c("P1", "m3", "M3", "P4", "P5", "P8", "M9")
dyad("c", x)
dyad("c", x, reverse = TRUE)
dyad("d e", "m3")
```

---

freq_ratio *Frequency ratios*

Description

Obtain frequency ratios data frame.

Usage

```
freq_ratio(x, ...)
```

Arguments

`x`	noteworthy or music object, or a numeric vector or list of numeric vectors for frequencies.
`...`	additional arguments: `ratios`, which is one of `"all"` (default), `"root"`, or `"range"` for filtering results. For frequency input, you may also specify `octaves` and `accidentals`. See details and examples.

Details

This generic function returns a data frame of frequency ratios from a vector or list of frequencies, a noteworthy object, or a music object. For frequency inputs, a list can be used to represent multiple timesteps. Octave numbering and accidentals are inferred from noteworthy and music objects, but can be specified for frequency. See examples.

By default ratios are returned for all combinations of intervals in each chord (`ratios = "all"`). `ratios = "root"` filters the result to only include chord ratios with respect to the root note of each chord. `ratios = "range"` filters to only the chord ratio between the root and highest note.

Value

a tibble data frame

Examples

```
x <- as_music("c4 e_ g ce_g")
(fr <- freq_ratio(x))

x <- music_notes(x)
identical(fr, freq_ratio(x))

x <- chord_freq(x)
identical(fr, freq_ratio(x))

freq_ratio(x, accidentals = "sharp")

freq_ratio(x, ratios = "root")

freq_ratio(x, ratios = "range")
```

---

guitarChords *Predefined guitar chords*

Description

A data frame containing information for many predefined guitar chords.

Usage

```
guitarChords
```

Format

A data frame with 12 columns and 3,967 rows

hp *Hammer ons and pull offs*

Description

Helper function for generating hammer on and pull off syntax.

Usage

```
hp(...)
```

Arguments

...	character, note durations. Numeric is allowed for lists of single inputs. See examples.

Details

This is especially useful for repeated instances. This function applies to general slur notation as well. Multiple input formats are allowed. Total number of note durations must be even because all slurs require start and stop points.

Value

character.

Examples

```
hp(16, 16)
hp("16 16")
hp("16 8 16", "8 16 8")
```

intervals *Interval helpers*

Description

Helper functions for musical intervals defined by two notes.

Usage

```
pitch_interval(notes1, notes2, use_root = TRUE)

pitch_diff(notes, use_root = TRUE, n = 1, trim = FALSE)

scale_interval(
  notes1,
  notes2,
  use_root = TRUE,
  format = c("mmp_abb", "mmp", "ad_abb", "ad")
)

scale_diff(
  notes,
  use_root = TRUE,
  n = 1,
  trim = FALSE,
  format = c("mmp_abb", "mmp", "ad_abb", "ad")
)

tuning_intervals(tuning = "standard")
```

Arguments

use_root	logical, use lowest pitch in chord for pitch intervals or scale intervals between adjacent timesteps. Otherwise intervals involving chords are NA.
notes, notes1, notes2	character, a noteworthy string. notes1 and notes2 must have equal number of timesteps.
n	integer, size of lag.
trim	logical, trim the n leading NA values from lagged intervals.
format	character, format of the scale notation: major/minor/perfect, augmented/diminished, and respective abbreviations. See argument options in defaults.
tuning	character, string tuning.

Details

Numeric intervals are directional. pitch_interval() returns the signed number of semitones defining the distance between two notes. Named scale intervals are names only. Use pitch for direction.

scale_interval() returns a character string that provides the named main interval, simple or compound, defined by the two notes. This function returns NA for any uncommon out of range large interval not listed as a named interval in mainIntervals().

pitch_interval() and scale_interval() compute intervals element-wise between two noteworthy strings. pitch_diff() and scale_diff() work similarly but compute lagged intervals on the elements in notes.

Value

a musical interval, integer or character depending on the function.

See Also

mainIntervals()

Examples

```
pitch_interval("b", "c4")
pitch_interval("c, e_, g_, a,", "e_, g_, a, c")
pitch_interval("c r", "dfa d")
pitch_interval("c r", "dfa d", use_root = FALSE)
scale_interval("c", "e_")
scale_interval("ceg", "egd'")

x <- "a, b, c d e f g# ac'e' a c' e'"
pitch_diff(x)
pitch_diff(x, use_root = FALSE)
scale_diff(x)
scale_diff(x, n = 2, trim = TRUE, use_root = FALSE)

# Lagged intervals respect rest timesteps.
# All timestep position including rests are retained.
# But the lag-n difference skips rest entries.
x <- "a, c r r r r g"
pitch_diff(x)
scale_diff(x)
pitch_diff(x, n = 2)
scale_diff(x, n = 2)
pitch_diff(x, n = 2, trim = TRUE)
scale_diff(x, n = 2, trim = TRUE)
```

---

interval_semitones *Interval semitones*

Description

Convert named intervals to numbers of semitones. For a complete list of valid interval names and abbreviations see mainIntervals(). interval may be a vector.

Usage

```
interval_semitones(interval)
```

Arguments

interval	character, interval ID. See details.

Value

integer

See Also

mainIntervals()

Examples

```
x <- c("minor third", "m3", "augmented second", "A2")
y <- c("P1", "m2", "M2", "m3", "M3", "P4", "TT", "P5")
interval_semitones(x)
interval_semitones(y)
```

---

is_diatonic *Check if notes and chords are diatonic*

Description

Check if notes and chords are diatonic in a given key.

Usage

```
is_diatonic(notes, key = "c")
```

Arguments

notes	character, a noteworthy string.
key	character, key signature.

Details

This function is a wrapper around is_in_scale(). To check if individual notes are in a scale, see note_in_scale().

Value

logical

See Also

is_in_scale()

Examples

```
is_diatonic("ceg ace ce_g", "c")
is_diatonic(c("r", "d", "dfa", "df#a"), "d")
```

keys *Key signatures*

Description

Helper functions for key signature information.

Usage

```
keys(type = c("all", "sharp", "flat"))

key_is_natural(key)

key_is_sharp(key)

key_is_flat(key)

key_n_sharps(key)

key_n_flats(key)

key_is_major(key)

key_is_minor(key)
```

Arguments

type	character, defaults to "all".
key	character, key signature.

Details

The keys() function returns a vector of valid key signature IDs. These IDs are how key signatures are specified throughout tabr, including in the other helper functions here via key. Like the other functions here, key_is_sharp() and key_is_flat() are for *key signatures*, not single pitches whose sharp or flat status is always self-evident from their notation. Major and minor keys are also self-evident from their notation, but key_is_major() and key_is_minor() can still be useful when programming.

Value

character vector.

Examples

```
keys()
key_is_natural(c("c", "am", "c#"))
x <- c("a", "e_")
key_is_sharp(x)
key_is_flat(x)
key_n_sharps(x)
key_n_flats(x)
```

`lilypond` *Save score to LilyPond file*

Description

Write a score to a LilyPond format (`.ly`) text file for later use by LilyPond or subsequent editing outside of R.

Usage

```
lilypond(
  score,
  file,
  key = "c",
  time = "4/4",
  tempo = "2 = 60",
  header = NULL,
  paper = NULL,
  string_names = NULL,
  endbar = "|.",
  midi = TRUE,
  colors = NULL,
  crop_png = TRUE,
  simplify = TRUE
)
```

Arguments

`score`	a score object.
`file`	character, LilyPond output file ending in `.ly`. May include an absolute or relative path.
`key`	character, key signature, e.g., `c`, `b_`, `f#m`, etc.
`time`	character, defaults to `"4/4"`.
`tempo`	character, defaults to `"2 = 60"`. Set to `NA` or `NULL` to suppress metronome mark in output. If suppressed and `midi = TRUE`, an error is thrown.
`header`	a named list of arguments passed to the header of the LilyPond file. See details.

paper	a named list of arguments for the LilyPond file page layout. See details.
string_names	label strings at beginning of tab staff. NULL (default) for non-standard tunings only, TRUE or FALSE for force on or off completely.
endbar	character, the global end bar.
midi	logical, add midi inclusion specification to LilyPond file.
colors	a named list of LilyPond element color overrides. See details.
crop_png	logical, alter template for cropped height. See details.
simplify	logical, uses simplify_phrase() to convert to simpler, more efficient LilyPond syntax.

Details

This function only writes a LilyPond file to disk. It does not require a LilyPond installation. It checks for the version number of an installation, but LilyPond is not required to be found.

This function can be used directly but is commonly used by render_* functions, which call this function internally to create the LilyPond file and then call LilyPond to render that file to sheet music.

Value

nothing returned; a file is written.

Header options

All header list elements are character strings. The options for header include the following.

- title
- subtitle
- composer
- album
- arranger
- instrument
- meter
- opus
- piece
- poet
- copyright
- tagline

Paper options

All paper list elements are numeric except page_numbers and print_first_page_number, which are logical. page_numbers = FALSE suppresses all page numbering. When page_numbers = TRUE, you can set print_first_page_number = FALSE to suppress printing of only the first page number. first_page_number is the number of the first page, defaulting to 1, and determines all subsequent page numbers. These arguments correspond to LilyPond paper block variables.

The options for paper include the following and have the following default values if not provided.

- textheight = 220
- linewidth = 150
- indent = 0
- fontsize = 10
- page_numbers = TRUE
- print_first_page_number = TRUE
- first_page_number = 1

PNG-related options

By default crop_png = TRUE. This alters the template so that when the LilyPond output file is created, it contains specifications for cropping the image to the content when that file is rendered by LilyPond to png. The image will have its width and height automatically cropped rather than retain the standard page dimensions. This only applies to png outputs made from the LilyPond file, not pdf. The argument is also ignored if explicitly providing textheight to paper. You may still provide linewidth to paper if you find you need to increase it beyond the default 150mm, generally as a result of using a large fontsize. Various render_* functions that wrap lilypond make use of this argument as well.

Color options

You can provide a named list of global color overrides for various sheet music elements with the colors argument of lilypond or one of the associated rendering functions.

By default, everything is black. Overrides are only inserted into the generated LilyPond file if given. Values are character; either the hex color or a named R color. The named list options include the following.

- color
- background
- staff
- time
- key
- clef
- bar
- beam
- head

- stem
- accidental
- slur
- tabhead
- lyrics

color is a global font color for the entire score. It affects staff elements and header elements. It does not affect everything, e.g., page numbers. background controls the background color of the entire page. Do not use this if making a transparent background png with the transparent argument available in the various render_* functions. The other options are also global but override color. You can change the color of elements broadly with color and then change the color of specific elements using the other options.

There are currently some limitations. Specifically, if you provide any background color override, most header elements will not display.

See Also

tab(), render_chordchart(), midily()

Examples

```
x <- phrase("c ec'g' ec'g'", "4 4 2", "5 432 432")
x <- track(x)
x <- score(x)
outfile <- file.path(tempdir(), "out.ly")
lilypond(x, outfile)
```

---

lilypond_root *LilyPond installation information*

Description

Details about local LilyPond installation and package API.

Usage

```
lilypond_root()

lilypond_version()

tabr_lilypond_api()
```

Details

Version information and installation directory are returned if the installation can be found. The LilyPond API references the currently loaded version of tabr.

Value

a message or system standard output.

Examples

```
lilypond_root()
lilypond_version()
tabr_lilypond_api()
```

`lp_chord_id` *LilyPond chord notation*

Description

Obtain LilyPond quasi-chord notation.

Usage

```
lp_chord_id(root, chord, exact = FALSE, ...)

lp_chord_mod(root, chord, exact = FALSE, ...)
```

Arguments

`root`	character, root note.
`chord`	character, `tabr` format chord name.
`exact`	logical, return a more exact LilyPond chord representation.
`...`	additional arguments passed to `transpose()`.

Details

These functions take a `tabr` syntax representation of a chord name and convert it to quasi-LilyPond syntax; "quasi" because the result still uses _ for flats and # for sharps, whereas LilyPond itself uses `es` and `is` (mostly). This is the format used by `tabr` functions involved in communicating with LilyPond for music transcription, and they make these final conversions on the fly. This can be overridden with `exact = TRUE`.

Value

character

Examples

```
lp_chord_id("a a a", "m M m7_5")
lp_chord_mod("a a a", "m M m7_5")
lp_chord_id("a a a", "m M m7_5", exact = TRUE)
lp_chord_mod("a a a", "m M m7_5", exact = TRUE)
```

lyrics *Create lyrics and check lyrics string validity*

Description

Functions for creating and checking lyrics objects.

Usage

```
lyrical(x)

as_lyrics(x, format = NULL)

is_lyrics(x)

lyrics_template(x, format = NULL)
```

Arguments

x	character or lyrics object. For `lyrics_template()`, an integer or one of the classes `noteworthy`, `noteinfo` or `music` to derive the number of timesteps from.
format	`NULL` or character, the timestep delimiter format, `"space"` or `"vector"`.

Details

The `lyrics` class is a simple class for arranging lyrics text by timestep. Its structure and behavior aligns with that of the classes `noteworthy`, `noteinfo` and `music`.

`lyrical()` is a trivial function that returns a scalar logical result essentially for any object that inherits from character, though this check may become more specific in the future.

`as_lyrics()` can be used to coerce to the `lyrics` class. Coercion will fail if the string is not lyrical. The `lyrics` class has its own `print()` and `summary()` methods.

When `format = NULL`, the timestep delimiter format is inferred from the lyrical string input.

Value

depends on the function

Examples

```
# space-delimited lyrics; use periods for timesteps with no lyric
x <- "These are the ly- rics . . . to this song"
is_lyrics(x)
lyrical(x)
as_lyrics(x)

# character vector; empty, period or NA for no lyric
x <- c("These", "are", "the", "ly-", "rics",
```

```
       "", ".", NA, "to", "this", "song") #
as_lyrics(x)

# generate empty lyrics object from noteworthy, noteinfo or music object
notes <- as_noteworthy("c d e d c r*3 e g c'")
x <- lyrics_template(notes)
x

x[1:5] <- strsplit("These are the ly- rics", " ")[[1]]
x[9:11] <- c("to", "this", "song")
x

summary(x)

attributes(x)
```

`mainIntervals` *Main musical intervals*

Description

A data frame containing descriptions of the main intervals, simple and compound.

Usage

```
mainIntervals
```

Format

A data frame with 5 columns and 26 rows

`midily` *Convert MIDI to LilyPond file*

Description

Convert a MIDI file (`.mid`) to a LilyPond format (`.ly`) text file.

Usage

```
midily(
  midi_file,
  file,
  key = "c",
  absolute = FALSE,
  quantize = NULL,
  explicit = FALSE,
```

```
  start_quant = NULL,
  allow_tuplet = c("4*2/3", "8*2/3", "16*2/3"),
  details = FALSE,
  lyric = FALSE
)
```

Arguments

midi_file	character, MIDI file (.mid). May include an absolute or relative path.
file	LilyPond output file ending in .ly.
key	key signature, defaults to "c".
absolute	logical, print absolute pitches (unavailable in current package version).
quantize	integer, duration, quantize notes on duration.
explicit	logical, print explicit durations.
start_quant	integer, duration, quantize note starts on the duration.
allow_tuplet	character vector, allow tuplet durations. See details.
details	logical, print additional information to console.
lyric	logical, treat all text as lyrics.

Details

Under development/testing. See warning and details below.

This function is a wrapper around the midi2ly() command line utility provided by LilyPond. It inherits all the limitations thereof. LilyPond is not intended to be used to produce meaningful sheet music from arbitrary MIDI files. While lilypond() converts R code score() objects to LilyPond markup directly, MIDI conversion to LilyPond markup by midily() requires LilyPond.

WARNING: Even though the purpose of the command line utility is to convert an existing MIDI file to a LilyPond file, it nevertheless generates a LilyPond file that *specifies inclusion of MIDI output*. This means when you subsequently process the LilyPond file with LilyPond or if you use miditab() to go straight from your MIDI file to pdf output, the command line tool will also produce a MIDI file output. It will overwrite your original MIDI file if it has the same file name and location!

allow_tuplets = NULL to disallow all tuplets. Fourth, eighth and sixteenth note triplets are allowed. The format is a character vector where each element is duration*numerator/denominator, no spaces. See default argument.

On Windows systems, it may be necessary to specify a path in tabr_options() to both midi2ly and python if they are not already added to the system PATH variable.

Value

nothing returned; a file is written.

See Also

miditab(), tab(), lilypond()

Examples

```
## Not run:
if(tabr_options()$midi2ly != ""){
  midi <- system.file("example.mid", package = "tabr")
  outfile <- file.path(tempdir(), "out.ly")
  midily(midi, outfile) # requires LilyPond installation
}

## End(Not run)
```

`miditab` *Convert MIDI to tablature*

Description

Convert a MIDI file to sheet music/guitar tablature.

Usage

```
miditab(midi_file, file, keep_ly = FALSE, details = FALSE, ...)
```

Arguments

`midi_file`	character, MIDI file (`.mid`). May include an absolute or relative path.
`file`	character, output file ending in .pdf or .png.
`keep_ly`	logical, keep LilyPond file.
`details`	logical, set to `TRUE` to print LilyPond log output to console. Windows only.
`...`	additional arguments passed to `midily()`.

Details

Under development/testing. See warning and details below.

Convert a MIDI file to a pdf or png music score using the LilyPond music engraving program. Output format is inferred from `file` extension. This function is a wrapper around `midily()`, the function that converts the MIDI file to a LilyPond (`.ly`) file using a LilyPond command line utility.

WARNING: Even though the purpose of the command line utility is to convert an existing MIDI file to a LilyPond file, it nevertheless generates a LilyPond file that *specifies inclusion of MIDI output*. This means when you subsequently process the LilyPond file with LilyPond or if you use `miditab()` to go straight from your MIDI file to pdf output, the command line tool will also produce a MIDI file output. It will overwrite your original MIDI file if it has the same file name and location!

On Windows systems, it may be necessary to specify a path in `tabr_options()` to both `midi2ly` and `python` if they are not already added to the system PATH variable.

Value

nothing returned; a file is written.

See Also

midily(), tab(), lilypond()

Examples

```
## Not run:
if(tabr_options()$midi2ly != ""){
  midi <- system.file("example.mid", package = "tabr")
  outfile <- file.path(tempdir(), "out.pdf")
  miditab(midi, outfile, details = FALSE) # requires LilyPond installation
}

## End(Not run)
```

---

mode-helpers *Mode helpers*

Description

Helper functions for working with musical modes.

Usage

```
modes(mode = c("all", "major", "minor"))

is_mode(notes, ignore_octave = FALSE)

mode_rotate(notes, n = 0, ignore_octave = FALSE)

mode_modern(
  mode = "ionian",
  key = "c",
  collapse = FALSE,
  ignore_octave = FALSE
)

mode_ionian(key = "c", collapse = FALSE, ignore_octave = FALSE)

mode_dorian(key = "c", collapse = FALSE, ignore_octave = FALSE)

mode_phrygian(key = "c", collapse = FALSE, ignore_octave = FALSE)

mode_lydian(key = "c", collapse = FALSE, ignore_octave = FALSE)

mode_mixolydian(key = "c", collapse = FALSE, ignore_octave = FALSE)

mode_aeolian(key = "c", collapse = FALSE, ignore_octave = FALSE)

mode_locrian(key = "c", collapse = FALSE, ignore_octave = FALSE)
```

Arguments

mode	character, which mode.
notes	character, for mode, may be a noteworthy string of seven notes, space- or vector-delimited.
ignore_octave	logical, strip octave numbering from modes not rooted on C.
n	integer, degree of rotation.
key	character, key signature.
collapse	logical, collapse result into a single string ready for phrase construction.

Details

For valid key signatures, see keys().

Modern modes based on major scales are available by key signature using the mode_* functions. The seven modes can be listed with modes. Noteworthy strings of proper length can be checked to match against a mode with is_mode(). Modes can be rotated with mode_rotate(), a wrapper around note_rotate().

Value

character

See Also

keys(), scale-helpers()

Examples

```
modes()
mode_dorian("c")
mode_modern("dorian", "c")
mode_modern("dorian", "c", ignore_octave = TRUE)

identical(mode_rotate(mode_ionian("c"), 1), mode_dorian("d"))
identical(
  mode_rotate(mode_ionian("c", ignore_octave = TRUE), 1),
  mode_dorian("d", ignore_octave = TRUE)
)

x <- sapply(modes(), mode_modern, ignore_octave = TRUE)
setNames(data.frame(t(x)), as.roman(1:7))
```

music *Create music objects and check music string validity*

Description

Check whether a string is comprised exclusively of valid syntax for music strings. A music object can be built from such a string. It combines a noteworthy string and a note info string.

Usage

```
musical(x)

as_music(
  notes,
  info = NULL,
  lyrics = NA,
  key = "c",
  time = "4/4",
  tempo = "2 = 60",
  accidentals = NULL,
  format = NULL,
  labels = NULL,
  at = seq_along(labels)
)

is_music(x)

music_split(x)
```

Arguments

x	character or music, a string to be coerced or an existing music object.
notes, info	noteworthy and note info strings. For as_music(), a complete music string is assumed for notes when info = NULL.
lyrics	optional lyrics object or NA, attached to output as an attribute.
key	character, store the key signature as a music attribute. Defaults to "c". See details.
time	character, store the time signature as a music attribute. Defaults to "4/4". See details.
tempo	character or NA, defaults to "2 = 60". See details.
accidentals	NULL or character, represent accidentals, "flat" or "sharp".
format	NULL or character, the timestep delimiter format, "space" or "vector".
labels	character, text annotations to attach to timesteps using notate.
at	integer, timesteps for labels, defaults to starting from time one.

Details

With note info strings, you are required to enter something at every timestep, even if it is only the duration. This makes sense because if you do not enter something, there is simply no indication of a timestep. A nice feature of music strings is that explicit timesteps are achieved just by having notes present, allowing you to leave out durations entirely when they repeat, inheriting them from the previous timestep where duration was given explicitly. There is no need to enter the same number across consecutive timesteps; the first will suffice and the rest are automatically filled in for you when the object is constructed.

`musical()` returns a scalar logical result indicating whether all timesteps contain exclusively valid entries.

`as_music()` can be used to coerce to the `music` class. Coercion will fail if the string is not musical. The `music` class has its own `print()` and `summary()` methods. `music` objects are primarily intended to represent an aggregation of a `noteworthy` object and a `noteinfo`. You can optionally fold in a `lyrics` object as well. However, for music data analysis, any operations will involve first splitting the object into its component parts. The value of this class is for the more efficient data entry it provides.

When `accidentals` or `format` are `NULL`, these settings are inferred from the musical string input. When mixed formats are present, flats are the default for accidentals.

Other attributes are attached to a `music` object. `key` uses the `tabr` syntax, e.g., `"c"`, `"b_"`, `"f#m"`, etc. `time` and `tempo` use the LilyPond string format. For music programming and analysis, `key`, `time` and `tempo` can most likely be ignored. They are primarily relevant when rendering a music snippet directly from a `music` object with LilyPond. These additional attributes provide more complete context for the rendered sheet music.

If you plan to render music snippets from a `music` object that you are defining from a new character string, and the context you have in mind is a stringed and fretted instrument like guitar, you can specify string numbers at the end of each timestep with numbers following a semicolon delimiter. This would still precede any `*` timestep multiplier number. See examples.

Note that if you convert a music object to a phrase object, you are changing contexts. The phrase object is the simplest LilyPond-format music structure. Coercion with `phrase()` strips all attributes of a music object and retains only notes, note info and string numbers.

Value

depends on the function

See Also

`music-helpers()`, `note-checks()`, `note-metadata()`, `note-summaries()`, `note-coerce()`

Examples

```
# note durations inherit from previous timestep if missing
x <- "a#4-+ b_[staccato] c,x d''t8( e)( g_')- a4 c,e_,g, ce_g4. a~8 a1"
is_music(x)
musical(x)
x <- as_music(x)
is_music(x)
```

```
x

y <- lyrics_template(x)
y[3:8] <- strsplit("These are some song ly- rics", " ")[[1]]
y

x <- as_music(x, lyrics = y, accidentals = "sharp")
summary(x)

# Starting string = 5: use ';5'. Carries over until an explicit change.
x <- "a,4;5*5 b,4-+ c4[staccato] cgc'e'~4 cgc'e'1 e'4;2 c';3 g;4 c;5 ce'1;51"
x <- as_music_df(as_music(x))
x$string
```

`music-helpers` *Accessing music object values and attributes*

Description

Helper functions for accessing music object values and attributes.

Usage

```
music_notes(x)

music_info(x)

music_strings(x)

music_key(x)

music_time(x)

music_tempo(x)

music_lyrics(x)
```

Arguments

`x`	music object.

Details

Note that while lyrics always shows as an attribute even when `NA`, `strings` is completely absent as a value if it was not part of the object construction from a new character string.

Value

depends on the function

See Also

music(), note-checks(), note-metadata(), note-summaries(), note-coerce()

Examples

```
# Starting string = 5: use ';5'. Carries over until an explicit change.
x <- "a,4;5*5 b,4- c4 cgc'e'~4 cgc'e'1 e'4;2 c';3 g;4 c;5 ce'1;51"
x <- as_music(x)

y <- lyrics_template(x)
y[3:8] <- strsplit("These are some song ly- rics", " ")[[1]]
y

x <- as_music(x, lyrics = y)

attributes(x)

music_split(x)

music_notes(x)
music_info(x)
music_key(x)
music_time(x)
music_tempo(x)
music_lyrics(x)
music_strings(x)
```

---

notate *Add text to music staff*

Description

Annotate a music staff, vertically aligned above or below the music staff at a specific note/time.

Usage

```
notate(x, text, position = "top")
```

Arguments

x	character.
text	character.
position	character, top or bottom.

Details

This function binds text annotation in LilyPond syntax to a note's associated `info` entry. Technically, the syntax is a hybrid form, but is later updated safely and unambiguously to LilyPond syntax with respect to the rest of the note info substring when it is fed to `phrase()` for musical phrase assembly.

Value

a character string.

Examples

```
notate("8", "Solo")
phrase("c'~ c' d' e'", pc(notate(8, "First solo"), "8 8 4."), "5 5 5 5")
```

`note-checks` *Basic noteworthy string checks*

Description

The simplest functions for inspecting noteworthy strings to see if their notes have certain properties.

Usage

```
note_is_accidental(notes)

note_is_natural(notes)

note_is_flat(notes)

note_is_sharp(notes)

note_has_accidental(notes)

note_has_natural(notes)

note_has_flat(notes)

note_has_sharp(notes)
```

Arguments

`notes`	character, a noteworthy string.

Details

Note that these functions are the weakest in terms of checking noteworthiness. They are simple regular expression-based wrappers. They are often used internally by more complex functions without wasting computational overhead on performing input validity checks, but they are exported from the package for user convenience. Their results will only make sense on strings that you define in accordance with noteworthy string rules.

The `note_is_*` functions return a logical vector with length equal to the number of timesteps in `notes`. The `note_has_*` functions summarize these to a single logical value.

Value

logical

See Also

note-metadata(), note-summaries(), note-coerce(), valid-notes()

Examples

```
x <- "r a_2 a a#' s"
note_has_accidental(x)
note_has_natural(x)
note_has_flat(x)
note_has_sharp(x)
note_is_accidental(x)
note_is_natural(x)
note_is_flat(x)
note_is_sharp(x)
note_has_tick(x)
note_has_integer(x)
note_is_tick(x)
note_is_integer(x)
note_has_rest(x)
note_is_rest(x)
```

---

note-coerce *Basic noteworthy strings formatting and coercion helpers*

Description

Helper functions for setting formatting attributes of noteworthy strings including representation of timesteps, octaves and accidentals.

Usage

```
naturalize(notes, type = c("both", "flat", "sharp"))

sharpen_flat(notes)

flatten_sharp(notes)

note_set_key(notes, key = "c")

as_tick_octaves(notes)

as_integer_octaves(notes)

as_space_time(x)
```

```
as_vector_time(x)

pretty_notes(notes, ignore_octave = TRUE)
```

Arguments

notes	character, a noteworthy string, space-delimited or vector of individual entries.
type	character, type of note to naturalize.
key	character, key signature to coerce any accidentals to the appropriate form for the key. May also specify "sharp" or "flat".
x	for generic functions: notes, info or music string.
ignore_octave	logical, strip any octave notation that may be present, returning only the basic notes without explicit pitch.

Details

For sharpen_flat() and flatten_sharp(), sharpening flats and flattening sharps refer to inverting their respective notation, not to raising or lowering a flatted or sharped note by one semitone. For the latter, use naturalize(), which removes flat and/or sharp notation from a string. note_set_key() is used for coercing a noteworthy string to a specific and consistent notation for accidentals based on a key signature. This is a wrapper around sharpen_flat() and flatten_sharp(). as_tick_octaves(), as_integer_octaves(), as_space_time() and as_vector_time() similarly affect octave and timestep format. For simultaneous control over the representation of timesteps, octave numbering and accidentals, all three are available as arguments to as_noteworthy().

Value

character

A note on generic functions

as_space_time() and as_vector_time() are generic since they apply clearly to and are useful for not only noteworthy strings, but also note info and music objects. If x is still a simple character string, these functions attempt to guess which of the three it is. It is recommended to set the class before using these functions.

There are many package functions that operate on noteworthy strings that could in concept work on music objects, but the expectation is that sound and time/info are disentangled. The music class is convenient for data entry, e.g., for transcription purposes, but it is not sensible to perform data analysis with quantities like pitch and time tightly bound together. This would only lead to repetitive deconstructions and reconstructions of music class objects. Most functions that operate on noteworthy strings or note info strings strictly apply to one or the other. Generic functions are reserved for only the most fundamental and generally applicable metadata retrieval and format coercion.

See Also

note-checks(), note-metadata(), note-summaries(), valid-notes()

Examples

```
x <- "e_2 a_, b_, c#f#a# c#'f#'a#''"
note_set_key(x, "f")
note_set_key(x, "g")
as_tick_octaves(x)
as_integer_octaves(x)
y <- as_vector_time(x)
is_vector_time(y)
is_space_time(as_space_time(y))

naturalize(x)
naturalize(x, "sharp")
sharpen_flat(x)
flatten_sharp(x)
pretty_notes(x)
```

---

note-equivalence *Note, pitch and chord equivalence*

Description

Helper functions to check the equivalence of two noteworthy strings, and other related functions.

Usage

```
note_is_equal(notes1, notes2, ignore_octave = TRUE)

note_is_identical(notes1, notes2, ignore_octave = TRUE)

pitch_is_equal(notes1, notes2)

pitch_is_identical(notes1, notes2)

octave_is_equal(notes1, notes2)

octave_is_identical(notes1, notes2, single_octave = FALSE)
```

Arguments

notes1	character, noteworthy string, space-delimited or vector of individual entries.
notes2	character, noteworthy string, space-delimited or vector of individual entries.
ignore_octave	logical, ignore octave position when considering equivalence.
single_octave	logical, for octave equality, require all notes share the same octave. See details.

Details

Noteworthy strings may contain notes, pitches and chords. Noteworthy strings are equal if they sound the same. This means that if one string contains Eb (e_) and the other contains D# (d#) then the two strings may be equal, but they are not identical.

`pitch_is_equal()` and `pitch_is_identical()` perform these respective tests of equivalence on both notes and chords. These are the strictest functions in terms of equivalent sound because pitch includes the octave number.

`note_is_equal()` and `note_is_identical()` are similar but include a default argument `ignore_octave = TRUE`, focusing only on the notes and chords. This allows an even more relaxed definition of equivalence. Setting this argument to `FALSE` is the same as calling the `pitch_is_*` variant.

Chords can be checked the same as notes. Every timestep in the sequence is checked pairwise between `note1` and `note2`.

These functions will return `TRUE` or `FALSE` for every timestep in a sequence. If the two noteworthy strings do not contain the same number of notes at a specific step, such as a single note compared to a chord, this yields a `FALSE` value, even in a case of an octave dyad with octave number ignored. If the two sequences have unequal length `NA` is returned. These are bare minimum requirements for equivalence. See examples.

`octave_is_equal()` and `octave_is_identical()` allow much weaker forms of equivalence in that they ignore notes completely. These functions are only concerned with comparing the octave numbers spanned by any pitches present at each timestep. When checking for equality, `octave_is_equal()` only looks at the octave number associated with the first note at each step, e.g., only the root note of a chord. `octave_is_identical()` compares all octaves spanned at a given timestep.

It does not matter when comparing two chords that they may be comprised of a different numbers of notes. If the set of unique octaves spanned by one chord is identical to the set spanned by the other, they are considered to have identical octave coverage. For example, `a1b2c3` is identical to `d1e1f2g3`. To be equal, it only matters that the two chords begin with `x1`, where `x` is any note. Alternatively, for `octave_is_identical()` only, setting `single_octave = TRUE` additionally requires that all notes from both chords being compared at a given timestep share a single octave.

Value

logical

Examples

```
x <- "b_2 ce_g"
y <- "b_ cd#g"
note_is_equal(x, y)
note_is_identical(x, y)

x <- "b_2 ce_g"
y <- "b_2 cd#g"
pitch_is_equal(x, y)
pitch_is_identical(x, y)

# same number of same notes, same order: unequal sequence length
x <- "b_2 ce_g b_"
y <- "b_2 ce_gb_"
```

```
note_is_equal(x, y)

# same number of same notes, order, equal length: unequal number per timestep
x <- "b_2 ce_g b_"
y <- "b_2 ce_ gb_"
note_is_equal(x, y)

x <- "a1 b_2 a1b2c3 a1b4 g1a1b1"
y <- "a_2 g#2 d1e1f2g3 a1b2b4 d1e1"
octave_is_equal(x, y)
octave_is_identical(x, y)
octave_is_identical(x, y, single_octave = TRUE)
```

---

note-logic *Relational operators for noteworthy class*

Description

Relational operators for comparing two noteworthy class objects.

Usage

```
## S3 method for class 'noteworthy'
e1 == e2

## S3 method for class 'noteworthy'
e1 != e2

## S3 method for class 'noteworthy'
e1 < e2

## S3 method for class 'noteworthy'
e1 <= e2

## S3 method for class 'noteworthy'
e1 > e2

## S3 method for class 'noteworthy'
e1 >= e2
```

Arguments

e1	noteworthy string.
e2	noteworthy string.

Details

Equality is assessed in the same manner as used for `note_sort()` when sorting pitches. What matters is the underlying semitone value associated with each pitch, not the string notation such as flat vs. sharp (see `pitch_is_identical()`). When comparing chords, or a chord vs. a single note, comparison favors the root. Comparison is made of the respective lowest pitches, then proceeds to the next pitch if equal.

For these operators, the objects on the left and right side of the operator must both be `noteworthy` or an error is returned.

The examples include a chord with its pitches entered out of pitch order. This does not affect the results because pitches within chords are sorted before note to note comparisons at each timestep are done between `e1` and `e2`.

Value

logical vector

Examples

```
x <- as_noteworthy("f# a d'f#'a' d'f#'a'")
y <- as_noteworthy("g_ b f#'a'd' d'd''")
x == y
x != y
x < y
x > y
x <= y
x >= y
```

---

note-metadata *Noteworthy string metadata*

Description

Inspect basic metadata for noteworthy strings.

Usage

```
n_steps(x)

n_notes(notes)

n_chords(notes)

n_octaves(notes)

chord_size(notes)

octave_type(notes)
```

```
accidental_type(x)

time_format(x)

is_space_time(x)

is_vector_time(x)

note_is_tick(notes)

note_is_integer(notes)

note_has_tick(notes)

note_has_integer(notes)

note_is_rest(notes)

note_has_rest(notes)
```

Arguments

x	for generic functions: notes, info or music string.
notes	character, a noteworthy string, space-delimited or vector of individual entries.

Details

These functions inspect the basic metadata of noteworthy strings. For functions that perform basic checks on strings, see note-checks().

The n_* functions give summary totals of the number of timesteps, number of individual note (non-chord) timesteps, number of chord time steps, and the number of distinct octaves present across timesteps.

Functions pertaining to type or format of a noteworthy string provide information on how a particular string is defined, e.g. time_format. Note that the result pertains to true noteworthy-class objects. If inspecting a standard character string, the result pertains to post-conversion to the noteworthy class and does not necessarily reflect what is found in notes verbatim. See examples.

Value

varies by function

A note on generic functions

n_steps() and the three time format functions are generic since they apply clearly to and are useful for not only noteworthy strings, but also note info, music, and lyrics objects. If x is still a simple character string, these functions attempt to guess if it is noteworthy, note info, or music. Lyrics content is arbitrary so is never considered for a simple character string. Best practice is to set the class before using these functions anyway.

There are many package functions that operate on noteworthy strings that could in concept also work on music objects, but the expectation is that sound and time/info are disentangled for analysis. The music class is convenient and relatively efficient data entry, e.g., for transcription purposes, but it is not sensible to perform data analysis with quantities like pitch and time tightly bound together in a single string. This would only lead to repetitive deconstructions and reconstructions of music class objects.

The music class is intended to be a transient class such as during data import, data entry, or data export. Most functions that operate on noteworthy strings or note info strings strictly apply to one or the other. Generic functions are reserved for only the most fundamental and generally applicable metadata retrieval and format coercion.

See Also

tabr-methods(), note-checks(), note-summaries(), note-coerce(), valid-notes()

Examples

```
x <- "e_2 a_, c#f#a#"
n_steps(x)
n_notes(x)
n_chords(x)
n_octaves(x)
chord_size(x)

# Type is mixed in `x` but is inferred under default conversion rules.
# These check `x` once validated and coerced to 'noteworthy' class.
octave_type(x)
accidental_type(x)
# The default is tick octaves and flats
as_noteworthy(x)

time_format(x)
is_space_time(x)
is_vector_time(x)
```

---

note-summaries *Noteworthy string summaries*

Description

Basic summary functions for noteworthy strings.

Usage

```
tally_notes(notes, rests = FALSE)

tally_pitches(notes, rests = FALSE)
```

```
octaves(notes)

tally_octaves(notes)

distinct_notes(notes, rests = FALSE)

distinct_pitches(notes, rests = FALSE)

distinct_octaves(notes)

pitch_range(notes)

semitone_range(notes)

semitone_span(notes)

octave_range(notes)

octave_span(notes)
```

Arguments

notes	character, a noteworthy string, space-delimited or vector of individual entries.
rests	logical, include rests r and silent rests s in tally.

Details

These functions provide basic summaries of noteworthy strings.

Returned object depends on the nature of the function. It can be integers, logical, character. Results can be a vector of equal length of a single value summary.

Use the `tally_*` and `distinct_*` functions specifically for summaries of unique elements.

`distinct_notes()` and `distinct_pitches()` filter a noteworthy string to its unique elements, respectively. These functions return another noteworthy string.
`*_span` functions are just the size of a range, e.g., `semitone_range()` and `semitone_span()`.

Value

varies by function

See Also

note-checks(), note-metadata(), note-coerce(), valid-notes()

Examples

```
x <- "r s e_2 a_, c#f#a#"
tally_notes(x)
tally_pitches(x)
octaves(x)
```

```
tally_octaves(x)
distinct_notes(x)
distinct_pitches(x)
distinct_octaves(x)

pitch_range(x)
semitone_range(x)
semitone_span(x)
octave_range(x)
octave_span(x)
```

`noteinfo` *Note info helpers*

Description

Functions for working with note info strings.

Usage

```
info_duration(x)

info_slur_on(x)

info_slur_off(x)

info_slide(x)

info_bend(x)

info_dotted(x)

info_single_dotted(x)

info_double_dotted(x)

info_annotation(x)

info_articulation(x)
```

Arguments

x	character, note info string normally accompanying a noteworthy string for building phrase objects. x may also be a phrase object.

Details

If x is a phrase object, there are some parsing limitations such as tuplets and repeats.

Value

character

See Also

valid-noteinfo()

Examples

```
a <- notate("t8x", "Start here")
notes <- "a, b, c d e f g# a r ac'e' a c' e' c' r*3 ac'e'~ ac'e'"
info <- paste(a, "t8x t8-. 16 4.. 16- 16 2^ 2 4. 8( 4)( 4) 8*4 1 1")
x <- as_music(notes, info)

data.frame(
  duration = info_duration(x),
  slur_on = info_slur_on(x),
  slur_off = info_slur_off(x),
  slide = info_slide(x),
  bend = info_bend(x),
  dotted = info_dotted(x),
  dotted1 = info_single_dotted(x),
  dotted2 = info_double_dotted(x),
  annotation = info_annotation(x),
  articulation = info_articulation(x)
)
```

---

note_ngram *Note/chord n-gram*

Description

Convert a noteworthy string to a list of noteworthy n-grams.

Usage

```
note_ngram(notes, n = 2, tally = FALSE, rests = FALSE)
```

Arguments

notes	a noteworthy string.
n	Number of grams. Must be >= 1 and <= number of timesteps in notes.
tally	logical, tally n-grams in a data frame. Otherwise a list.
rests	logical, exclude rests. Affects the number of timesteps.

Value

list of noteworthy objects or a tibble

Examples

```
x <- as_noteworthy("c r ceg dfa ceg dfa")
note_ngram(x)
(x <- note_ngram(x, tally = TRUE))
x$ngram <- as.character(x$ngram)
x
```

`note_slice` *Slice, sort, rotate, shift and arpeggiate notes*

Description

Helper functions for indexing and moving notes within noteworthy strings.

Usage

```
note_slice(notes, ...)

note_sort(notes, decreasing = FALSE)

note_rotate(notes, n = 0)

note_shift(notes, n = 0)

note_arpeggiate(notes, n = 0, step = 12)
```

Arguments

`notes`	character, a noteworthy string, space-delimited or vector of individual entries.
`...`	For `note_slice()`, an integer or logical vector.
`decreasing`	logical, short in decreasing order.
`n`	integer, number of rotations or extensions of note sequence. See details.
`step`	integer, number of semitone steps from the first (or last) note in `notes` at which to begin repeating the shifted `notes` sequence as an arpeggio. See examples.

Details

`note_slice()` subsets the timesteps of a noteworthy string by integer index or logical vector of length equal to the number of timesteps.

`note_sort()` sorts the timesteps of a noteworthy string by pitch. When a tie exists by root note, the next note in chords are compared, if they exist. For example, `a,` sorts lower than `a,ce`.

`note_rotate()` simply rotates anything space-delimited or vectorized in place. It allows chords. Octave numbering is ignored if present.

For `note_shift()` the entire sequence is shifted up or down in pitch, as if inverting a broken chord. If `notes` contains chords, they are broken into successive notes. Then all notes are ordered by pitch. Finally shifting occurs.

Instead of a moving window, note_arpeggiate() grows its sequence from the original set of timesteps by repeating the entire sequence n times (n must be positive). Each repeated sequence contributing to the arpeggio is offset by step semitones from the original. step can be negative. It defaults to 12, increasing all notes by one octave.

Value

character

Examples

```
x <- "bd'f#' a c'e'g' b ba c'g' gd'g'd''"
note_sort(x)
note_sort(x, decreasing = TRUE)

x <- "e_2 a_, c#f#a#"
note_slice(x, 2:3)
note_slice(x, c(FALSE, TRUE, TRUE))

note_rotate(x, 1)

note_shift("c e g", 1)
note_shift("c e g", -4)

note_arpeggiate("c e g ceg", 3)
note_arpeggiate("c e g", 3, step = -12)
note_arpeggiate("g e c", 3, step = -12)
note_arpeggiate("c e_ g_ a", 3, step = 3)
note_arpeggiate("c a g_ e_", 3, step = -3)
```

---

n_measures *Summarize rhythm and time of music objects*

Description

These functions assist with summarizing temporal data for music objects.

Usage

```
n_measures(x)

n_beats(x, unit = 4)

steps_per_measure(x)

bpm(x, unit = 4, tempo = NULL)

seconds(x, tempo = NULL)
```

```
seconds_per_measure(x, tempo = NULL)

seconds_per_step(x, tempo = NULL)

steps_start_time(x, tempo = NULL)
```

Arguments

x	note info or music object.
unit	character, or an equivalent integer. A beat unit. See details.
tempo	character, LilyPond format tempo, e.g., "4 = 120" is 120 quarter note beats per minute.

Details

These functions also work with the simpler `noteinfo` class, though some functions require you to provide additional arguments.

Functions that deal with real time require a known tempo, which music objects have. The simpler note info object does not contain this information. You can provide a value to the `tempo` argument of such functions. This overrides the tempo of `x` if a music object. But the reason to use `tempo` is to provide one when `x` is a note info object. By default `tempo = NULL`, in which case it will derive the value from the music object or return an error for note info objects.

`n_measures()` gives the total number of measures covered by all timesteps. Functions providing the number of beats and beats per minute both take a `unit`, defaulting to 4 for quarter note beats. The unit can be any even beat, triplet beat, dotted, or double dotted beat, from `"t32"` up to 1.

The number of timesteps starting in each measure is obtained with `steps_per_measure()`.

Value

depends on function

Examples

```
a <- notate("t8x", "Start here")
notes <- "a, b, c d e f g# a r ac'e' a c' e' c' r*3 ac'e'~ ac'e'"
info <- paste(a, "t8x t8-. 16 4.. 16- 16 2^ 2 4. 8( 4)( 4) 8*4 1 1")
info <- as_noteinfo(info)
x <- as_music(notes, info)

n_measures(info) # fraction indicates incomplete final measure
n_measures(x)

n_beats(x)
n_beats(x, 1)
n_beats(x, "t16")

bpm(x)
bpm(x, "t8")
```

```
seconds(x)
seconds(info, "4 = 120")
seconds(info, "2 = 60")
seconds(x, "4 = 100")

steps_per_measure(x)
seconds_per_measure(x)
seconds_per_step(x)
steps_start_time(x)
```

`phrase` *Create a musical phrase*

Description

Create a musical phrase from character strings that define notes, note metadata, and optionally explicit strings fretted. The latter can be used to ensure proper tablature layout.

Usage

```
phrase(notes, info = NULL, string = NULL, bar = NULL)

p(notes, info = NULL, string = NULL, bar = NULL)
```

Arguments

`notes`, `info`	noteworthy and note info strings. When `info = NULL`, it is assumed that `notes` refers to a music object or string formatted as such.
`string`	space-delimited character string or vector (or integer vector if simple string numbers). This is an optional argument that specifies which instrument strings to play for each specific timestep. Otherwise `NULL`.
`bar`	character or `NULL` (default). Terminates the phrase with a bar or bar check. See details. Also see the LilyPond help documentation on bar notation for all the valid options.

Details

A phrase object combines a valid string of notes with a corresponding valid string of note info. The only required note info is time, but other information can be included as well. You do not need to input an existing `noteworthy` class object and `noteinfo` class object, but both inputs must be valid and thus coercible to these classes. This is similar to how the `music` class works. The difference with phrase objects is that they are used to create LilyPond syntax analogous to what a music object contains.

Note that if you convert a music object to a phrase object, you are changing contexts. The phrase object is the simplest LilyPond-format music structure. Coercion with `phrase()` strips all attributes of a music object and retains only notes, note info and string numbers.

See the help documentation on noteworthy, noteinfo, and music classes for an understanding of the input data structures. The function p() is a convenient shorthand wrapper for phrase().

If a string is provided to bar, it is interpreted as LilyPond bar notation. E.g., bar = "|" adds the LilyPond syntax \bar "|" to the end of a phrase. If only a bar check is desired, use bar = TRUE. FALSE is treated as NULL for completeness.

Value

a phrase.

See Also

valid-notes(), valid-noteinfo(), music()

Examples

```
phrase("c ec'g' ec'g'", "4- 4 2") # no string arg (not recommended for tabs)
phrase("c ec4g4 ec4g4", "4 4 2") # same as above
phrase("c b, c", "4. 8( 8)", "5 5 5") # direction implies hammer on
phrase("b2 c d", "4( 4)- 2", "5 5 5") # hammer and slide

phrase("c ec'g' ec'g'", "1 1 1", "5 432 432")
p("c ec'g' ec'g'", 1, "5 4 4") # same as above


n <- "a, b, c d e f g e f g a~ a"
i <- "8- 8 8 8-. t8( t8)( t8) t16( t16)( t16) 8 1"
m <- as_music(n, i)

x <- p(n, i)
x
identical(x, p(m))

x <- "a,4;5*5 b,4- c4 cgc'e'~4 cgc'e'1 e'4;2 c';3 g;4 c;5 ce'1;51"
p(x)
identical(p(x), p(as_music(x)))

x <- p("a b", 2, bar = "|.")
x2 <- pc(p("a b", 2), '\\bar "|."')
identical(x, x2)
```

---

phrase-checks *Phrase validation and coercion*

Description

These helper functions add some validation checks for phrase and candidate phrase objects.

Usage

```
as_phrase(phrase)

phrasey(phrase)

notify(phrase)

phrase_notes(phrase, collapse = TRUE)

phrase_info(phrase, collapse = TRUE, annotations = TRUE)

phrase_strings(phrase, collapse = FALSE)

notable(phrase)
```

Arguments

phrase	phrase object or character string (candidate phrase).
collapse	logical, collapse result into a single string ready for phrase construction.
annotations	logical, strip any text annotations from the note info converted from phrase().

Details

Use these functions with some caution. They are not intended for strictness and perfection. phrasey() checks whether an object is weakly phrase-like and returns TRUE or FALSE. It can be used to safeguard against the most obvious cases of phrase() not containing valid phrase syntax when programming. However, it may also be limiting. Use wear sensible.

as_phrase() coerces an object to a phrase object if possible. This function performs an internal phrasey() check.

notify() attempts to decompose a phrase object back to its original input vectors consisting of notes, note info, and optionally, instrument string numbering. If successful, it returns a tibble data frame with columns: notes, info, string.

Unless decomposing very simple phrases, this function is likely to reveal limitations. Complex phrase objects constructed originally with phrase() can be challenging to deconstruct in a one to one manner. Information may be lost, garbled, or the function may fail. For example, this function is not advanced enough to unravel repeat notation or tuplets.

notable() returns TRUE or FALSE regarding whether a phrase can be converted back to character string inputs, not necessarily with complete correctness, but without simple failure.It checks for phrasiness. Then it tries to call notify() and returns FALSE gracefully if that call throws an exception.

Value

see details for each function's purpose and return value.

Examples

```
# Create a list of phrase objects
p1 <- phrase("c ec'g' ec'g'", "4 4 2") # no string numbers (not recommended)
p2 <- phrase("c ec4g4 ec4g4", "4 4 2") # same as above
p3 <- phrase("c b, c", "4. 8( 8)", "5 5 5") # direction implies hammer on
p4 <- phrase("b2 c d", "4( 4)- 2", "5 5 5") # hammer and slide
p5 <- phrase("c ec'g'~ ec'g'", 1, "5 432 432") # tied chord
x <- list(p1, p2, p3, p4, p5)

# Check if phrases and strings are phrasey
sapply(x, phrasey)
sapply(as.character(x), phrasey, USE.NAMES = FALSE)

# Coerce character string representation to phrase and compare with original
y <- lapply(as.character(x), as_phrase)
identical(x, y)

# Check if notable
sapply(x, notable)
notable(p("a b c", 1))
notable("a b x") # note: not constructible as a phrase in the first place

# Notify phrases
d <- do.call(rbind, lapply(x, notify))
d

# Wrappers around notify extract components, default to collapsed strings
phrase_notes(p5)
phrase_info(p5)
phrase_strings(p5)

# If phrase decomposition works well, coercion is one to one
x2 <- lapply(x,
  function(x) p(phrase_notes(x), phrase_info(x), phrase_strings(x))
)
identical(x, x2)
```

`pitch_freq` *Pitch conversions*

Description

Convert between pitches, chords, semitones and frequencies.

Usage

```
pitch_freq(notes, a4 = 440)

pitch_semitones(notes)
```

```
chord_freq(notes, a4 = 440)

chord_semitones(notes)

freq_pitch(
  freq,
  octaves = c("tick", "integer"),
  accidentals = c("flat", "sharp"),
  collapse = FALSE,
  a4 = 440
)

freq_semitones(freq, a4 = 440)

semitone_pitch(
  semitones,
  octaves = c("tick", "integer"),
  accidentals = c("flat", "sharp"),
  collapse = FALSE
)

semitone_freq(semitones, a4 = 440)
```

Arguments

notes	character, noteworthy string, space-delimited or vector of individual entries. See details.
a4	the fixed frequency of the A above middle C, typically 440 Hz.
freq	numeric vector, frequencies in Hz.
octaves	NULL or character, "tick" or "integer" octave numbering in result.
accidentals	NULL or character, represent accidentals, "flat" or "sharp".
collapse	logical, collapse result into a single string. key and style.
semitones	integer values of pitches.

Details

Frequencies are in Hertz. Values are based on the 12-tone equal-tempered scale. When converting an arbitrary frequency to pitch, it is rounded to the nearest pitch. pitch_freq() and pitch_semitones() strictly accept single notes in noteworthy strings and return numeric vectors. chord_freq() and chord_semitones() accept any noteworthy string and always return a list. These are provided so that all functions are type-safe. See examples.

Value

integer, numeric or noteworthy vector

Examples

```
x <- "a e4 a4 e5 a5"
y <- pitch_freq(x)
y

freq_semitones(y)
freq_pitch(y)

identical(as_noteworthy(x), freq_pitch(y, "integer", collapse = TRUE))

s <- pitch_semitones(x)
s
semitone_pitch(s)

x <- "a, a,c#e"
chord_semitones(x)
chord_freq(x)
```

`pitch_seq` *Create a sequence from pitch notation*

Description

Create a noteworthy string of a sequence of consecutive pitches.

Usage

```
pitch_seq(x, y, key = NULL, scale = NULL, format = c("space", "vector"))
```

Arguments

`x`	character, valid pitch notation, e.g., `"a2"` or `"a,"`.
`y`	character, same as `x` for the sequence `x:y`. If a number, the length of the sequence from `x` and the sign of `y` determines the direction.
`key`	character, key signature for a diatonic sequence. `key = NULL` (default) results in a chromatic sequence.
`scale`	character, if you want to use a different scale in conjunction with the key/root note, you can provide it, e.g., `scale = "harmonic minor"`. Ignored if `key = NULL`.
`format`	character, the timestep delimiter format, `"space"` or `"vector"`.

Details

Note that all pitches resulting from the defined sequence must be in the semitone range 0-131 or an error is thrown.

If not using a chromatic sequence and `x` (or `y` if also a pitch) is not part of the key signature or scale, the sequence is internally bound. See examples.

Format of accidentals in the result is prioritized by the scale and key, the key when no scale is given, then x (and y if also a pitch), and finally defaults to flats if ambiguous.

Value

noteworthy

Examples

```
# chromatic sequence (default)
pitch_seq("a,", 13)
pitch_seq("c5", -2)
pitch_seq("c", "b")

# diatonic sequence
pitch_seq("c", 8, key = "c")
pitch_seq("c", 8, "am")
pitch_seq("c#,", "a#'", "am")

# combine with alternative scale
pitch_seq("a", 8, "am", "harmonic minor")
```

---

plot_fretboard *Chord and fretboard diagram plots*

Description

Create a fretboard diagram for a single chord or a general progression.

Usage

```
plot_fretboard(
  string,
  fret,
  labels = NULL,
  mute = FALSE,
  label_size = 10,
  label_color = "white",
  point_size = 10,
  point_color = "black",
  point_fill = "black",
  group = NULL,
  horizontal = FALSE,
  left_handed = FALSE,
  fret_range = NULL,
  fret_labels = NULL,
  fret_offset = FALSE,
  accidentals = c("flat", "sharp"),
```

```
  tuning = "standard",
  show_tuning = FALSE,
  asp = NULL,
  base_size = 20
)

plot_chord(
  chord,
  labels = NULL,
  label_size = 10,
  label_color = "white",
  point_size = 10,
  point_color = "black",
  point_fill = "black",
  group = NULL,
  horizontal = FALSE,
  left_handed = FALSE,
  fret_range = NULL,
  fret_labels = NULL,
  fret_offset = FALSE,
  accidentals = c("flat", "sharp"),
  tuning = "standard",
  show_tuning = FALSE,
  asp = NULL,
  base_size = 20
)
```

Arguments

string	integer or as a space-delimited character string; instrument string numbers.
fret	integer or as a space-delimited character string; fret numbers.
labels	NULL or character, optional vector of text labels, must be one for every point; or just the special value "notes".
mute	logical vector or specific integer indices, which notes to mute. See details.
label_size	numeric, size of fretted note labels.
label_color	character, label color.
point_size	numeric, size of fretted note points.
point_color	character, point color.
point_fill	character, point fill color.
group	optional vector to facet by.
horizontal	logical, directional orientation.
left_handed	logical, handedness orientation.
fret_range	fret limits, if not NULL, overrides limits derived from fret.
fret_labels	integer, vector of fret number labels for fret axis. See details.

fret_offset	logical set to TRUE to shift the fret axis number labels (if present) from being directly next to the fret to being aligned with the circles behind the fret.
accidentals	character, when labels = "notes" represent accidentals: "flat" or "sharp".
tuning	explicit tuning, e.g., "e, a, d g b e'", or a pre-defined tuning. See details.
show_tuning	logical, show tuning of each string on string axis.
asp	numeric, aspect ratio, overrides default aspect ratio derived from number of strings and frets.
base_size	base size for ggplot2::theme_void().
chord	character, a single chord given in fret notation. See details.

Details

These functions are under development and subject to change. They each return a ggplot object.

Use plot_chord() to create a fretboard diagram of a specific chord. plot_chord() accepts a character string in simple fretboard format, e.g., chord = "xo221o". Zero is allowed in place of "o". This only works when no spaces or semicolons are detected. The function checks for spaces first, then semicolons, to split fret numbers. Do not mix formats. For example, you can use chord = "xo221o", chord = "x 8 10 10 9 8" or chord = "x;8;10;10;9;8". Trailing delimiters are ignored (LilyPond format: "x;8;10;10;9;8;"). If there are fewer fret values than there are strings on the instrument, as inferred from tuning, then muted strings, x, are inferred for the remaining lower-pitch strings.

plot_fretboard() produces a more general fretboard diagram plot. It is intended for scales, arpeggios and other patterns along the fretboard. For this function, provide vectors of string and fret numbers. mute is available but not as applicable for this function; it is a pass-through from plot_chord(). For single chord diagrams, use plot_chord(). The letter "o" is also allowed in fret for open strings and will display below the lowest fret plotted. The number 0 is treated with the intent of displaying the corresponding position on the instrument neck.

Number of strings is derived from tuning. See tunings() for pre-defined tunings and examples of explicit tunings. tuning affects point labels when labels = "notes".

Providing fret_labels overrides the default (minimal) fret numbering behavior for the fret axis. These are only intended to be integers. The vector of integers given is sorted and subset if needed to the range of frets that appear in the plot. See example.

Value

a ggplot object

Examples

```
# General patterns: scale shifting exercise
string <- c(6, 6, 6, 5, 5, 5, 4, 4, 4, 4, 4, 3, 3, 3, 2, 2, 2, 1, 1, 1)
fret <- "2 4 5 2 4 5 2 4 6 7 9 6 7 9 7 9 10 7 9 10" # string input accepted
plot_fretboard(string, fret, labels = "notes", fret_offset = TRUE)
plot_fretboard(string, fret, fret_labels = c(3, 5, 7, 9, 12), show_tuning = TRUE)

# open and muted strings on shifted general fretboard layout
# try to use plot_chord() if more suitable
```

```
plot_fretboard("6 5 4 3", "o 9 10 12", mute = 2, show_tuning = TRUE)

# Single chord diagrams
# open chord
idx <- c(1, 1, 2, 2, 2, 1)
fill <- c("white", "black")[idx]
lab_col <- c("black", "white")[idx]
plot_chord("xo221o", "notes", label_color = lab_col, point_fill = fill)

# moveable chord
plot_chord("355433", horizontal = TRUE, show_tuning = TRUE)

# leading x inferred; same as plot_chord("xxo321")
plot_chord("o231", fret_labels = 3)
plot_chord("10 12 13 11", show_tuning = TRUE)
plot_chord("o x 10 12 13 11", fret_range = c(9, 14), fret_labels = c(9, 12))
```

`plot_music` *Plot sheet music snippet with LilyPond*

Description

These functions are wrappers around the `render_music*` functions. They abstract the process of rendering a sheet music snippet to png and loading the rendered image back into R to be displayed as a plot in an open graphics device or inserted into an R Markdown code chunk.

Usage

```
plot_music(
  music,
  clef = "treble",
  tab = FALSE,
  tuning = "standard",
  string_names = NULL,
  header = NULL,
  paper = NULL,
  colors = NULL,
  transparent = FALSE,
  res = 300
)

plot_music_tc(
  music,
  header = NULL,
  paper = NULL,
  colors = NULL,
  transparent = FALSE,
  res = 300
```

```
)

plot_music_bc(
  music,
  header = NULL,
  paper = NULL,
  colors = NULL,
  transparent = FALSE,
  res = 300
)

plot_music_tab(
  music,
  clef = NA,
  tuning = "standard",
  string_names = NULL,
  header = NULL,
  paper = NULL,
  colors = NULL,
  transparent = FALSE,
  res = 300
)

plot_music_guitar(
  music,
  tuning = "standard",
  string_names = NULL,
  header = NULL,
  paper = NULL,
  colors = NULL,
  transparent = FALSE,
  res = 300
)

plot_music_bass(
  music,
  tuning = "bass",
  string_names = FALSE,
  header = NULL,
  paper = NULL,
  colors = NULL,
  transparent = FALSE,
  res = 300
)
```

Arguments

music	a music object.

clef	character, include a music staff with the given clef. NA to suppress. See track() for details.
tab	logical, include tablature staff. NA to suppress. See track().
tuning	character, string tuning, only applies to tablature. See track().
string_names	label strings at beginning of tab staff. NULL (default) for non-standard tunings only, TRUE or FALSE for force on or off completely.
header	a named list of arguments passed to the header of the LilyPond file. See lilypond() details.
paper	a named list of arguments for the LilyPond file page layout. See lilypond() details.
colors	a named list of LilyPond element color global overrides. See lilypond() for details.
transparent	logical, transparent background for intermediate png file.
res	numeric, resolution, png only. Defaults to 300.

Details

While these functions abstract away the details of the process, this is not the same as making the plot completely in R. R is only displaying the intermediary png file. LilyPond is required to engrave the sheet music.

For R Markdown you can alternatively render the png using the corresponding render_music* function and then place it in the document explicitly using knitr::include_graphics(). See render_music() for more details.

Value

a plot

See Also

render_music(), phrase(), track(), score(), lilypond(), tab()

Examples

```
x <- "a,4;5*5 b,4- c4 cgc'e'~4 cgc'e'1 e'4;2 c';3 g;4 c;5 ce'1;51"
x <- as_music(x)

y <- "a,,4;3*5 b,,4- c,4 c,g,c~4 c,g,c1 c4;1 g,,;2 c,,;3 g,,;2 c,c1;31"
y <- as_music(y)

## Not run:
if(tabr_options()$lilypond != ""){ # requires LilyPond installation
  plot_music(x)
  plot_music(x, "treble_8", tab = TRUE)

  plot_music_tc(x)
  plot_music_bc(x)
```

```
  plot_music_tab(x)
  plot_music_guitar(x)
  plot_music_bass(y)
}

## End(Not run)
```

`ratio_to_cents` *Convert between chord frequency ratios and cents*

Description

Convert between frequency ratios and logarithmic cents

Usage

```
ratio_to_cents(x, y = NULL)

cents_to_ratio(x)
```

Arguments

x	a vector of ratios if y = NULL, otherwise frequencies. Cents for cents_to_ratio().
y	if not NULL, frequencies and the ratios are given by y / x.

Value

numeric

Examples

```
ratio_to_cents(c(0.5, 1, 1.5, 2))
cents_to_ratio(c(-1200, 0, 701.955, 1200))
```

`read_midi` *Read, inspect and convert MIDI file contents*

Description

Read MIDI file into a data frame and inspect the music data with supporting functions.

Usage

```
read_midi(file, ticks_per_qtr = 480)

midi_metadata(x)

midi_notes(x, channel = NULL, track = NULL, noteworthy = TRUE)

midi_time(x)

midi_key(x)

ticks_to_duration(x, ticks_per_qtr = 480)

duration_to_ticks(x, ticks_per_qtr = 480)
```

Arguments

file	character, path to MIDI file.
ticks_per_qtr	ticks per quarter note. Used to compute durations from MIDI file ticks.
x	a data frame returned by read_midi(). An integer vector for ticks_to_duration(); a character vector (may be a space-delimited string) for duration_to_ticks().
channel, track	integer, filter rows on channel or track.
noteworthy	logical, convert to noteworthy and noteinfo data.

Details

The read_midi() function wraps around tuneR::readMidi() by Uwe Ligges and Johanna Mielke. midi_notes() is a work in progress, but converts MIDI data to noteworthy strings and note info formats. This makes it easy to analyze, transform and edit the music data as well as render it to sheet music and a new MIDI file.

read_midi() does not parse the ticks per quarter note from the MIDI file input at this time. It must be specified with ticks_per_qtr.

Value

a tibble data frame

Examples

```
ticks_to_duration(c(120, 160))
ticks_to_duration(c(128, 192, 512), ticks_per_qtr = 384)
duration_to_ticks(c("t8", "8", "8.", "8.."))
duration_to_ticks(c("t8 8 8. 8.."), ticks_per_qtr = 384)

file <- system.file("example2.mid", package = "tabr")
if(require("tuneR")){
  x <- read_midi(file, ticks_per_qtr = 384)
  midi_metadata(x)
```

```
  midi_time(x)
  midi_key(x)
  midi_notes(x, channel = 0, noteworthy = FALSE)

  (x <- midi_notes(x, channel = 0))
  (x <- as_music(x$pitch, x$duration))

  # requires LilyPond installation
  if(tabr_options()$lilypond != ""){
    out <- file.path(tempdir(), "out.pdf")
    phrase(x) |> track_bc() |> score() |> tab(out, details = FALSE)
  }
}
```

`render_chordchart` *Render a chord chart with LilyPond*

Description

Render a standalone chord chart of chord fretboard diagrams with LilyPond for a set of chords.

Usage

```
render_chordchart(
  chords,
  file,
  size = 1.2,
  header = NULL,
  paper = NULL,
  colors = NULL,
  crop_png = TRUE,
  transparent = FALSE,
  res = 150,
  keep_ly = FALSE,
  details = FALSE
)
```

Arguments

`chords`	named character vector of valid formatting for LilyPond chord names and values. See examples.
`file`	output file.
`size`	numeric, size of fretboard diagrams (relative to paper font size). Use this to scale diagrams up or down.
`header`	a named list of arguments passed to the header of the LilyPond file. See details.
`paper`	a named list of arguments for the LilyPond file page layout. See details.
`colors`	reserved; not yet implemented for this function.

crop_png	logical, see lilypond() for details.
transparent	logical, transparent background, png only.
res	numeric, resolution, png only. transparent = TRUE may fail when res exceeds ~150.
keep_ly	logical, keep intermediate LilyPond file.
details	logical, set to TRUE to print LilyPond log output to console. Windows only.

Details

This function uses a generates a LilyPond template for displaying only a fretboard diagram chart. It then passes the file to LilyPond for rendering. To plot specific fretboard diagrams in R using ggplot and with greater control, use plot_fretboard().

The options for paper include the following and have the following default values if not provided.

- textheight = 220
- linewidth = 150
- indent = 0
- fontsize = 10
- page_numbers = FALSE
- print_first_page_number = TRUE
- first_page_number = 1

fontsize only controls the global font size. If you want to scale the size of the fretboard diagrams up or down use the the size argument rather than this paper value.

Note that chord chart output must fit on a single page. If the full set of chord diagrams does not fit on one page then diagrams will be clipped in the rendered output. Use size to keep the output to one page or make multiple sheets separately.

Value

writes files to disk

See Also

plot_fretboard(), lilypond(), tab()

Examples

```
suppressPackageStartupMessages(library(dplyr))

chords <- filter(
  guitarChords, root %in% c("c", "f") & id %in% c("7", "M7", "m7") &
  !grepl("#", notes) & root_fret <= 12) |>
  arrange(root, id)
chords <- setNames(chords$fretboard, chords$lp_name)
head(chords)
```

```
# requires LilyPond installation
if(tabr_options()$lilypond != ""){
  outfile <- file.path(tempdir(), "out.pdf")
  hdr <- list(
    title = "Dominant 7th, major 7th and minor 7th chords",
    subtitle = "C and F root"
  )
  render_chordchart(chords, outfile, 2, hdr, list(textheight = 175))
}
```

---

render_music *Render sheet music snippet with LilyPond*

Description

Render a sheet music/tablature snippet from a music object with LilyPond.

Usage

```
render_music(
  music,
  file,
  clef = "treble",
  tab = FALSE,
  tuning = "standard",
  string_names = NULL,
  header = NULL,
  paper = NULL,
  midi = FALSE,
  colors = NULL,
  transparent = FALSE,
  res = 150,
  keep_ly = FALSE,
  simplify = TRUE
)

render_music_tc(
  music,
  file,
  header = NULL,
  paper = NULL,
  midi = FALSE,
  colors = NULL,
  transparent = FALSE,
  res = 150,
  keep_ly = FALSE,
  simplify = TRUE
)
```

```
render_music_bc(
  music,
  file,
  header = NULL,
  paper = NULL,
  midi = FALSE,
  colors = NULL,
  transparent = FALSE,
  res = 150,
  keep_ly = FALSE,
  simplify = TRUE
)

render_music_tab(
  music,
  file,
  clef = NA,
  tuning = "standard",
  string_names = NULL,
  header = NULL,
  paper = NULL,
  midi = FALSE,
  colors = NULL,
  transparent = FALSE,
  res = 150,
  keep_ly = FALSE,
  simplify = TRUE
)

render_music_guitar(
  music,
  file,
  tuning = "standard",
  string_names = NULL,
  header = NULL,
  paper = NULL,
  midi = FALSE,
  colors = NULL,
  transparent = FALSE,
  res = 150,
  keep_ly = FALSE,
  simplify = TRUE
)

render_music_bass(
  music,
  file,
```

```
  tuning = "bass",
  string_names = NULL,
  header = NULL,
  paper = NULL,
  midi = FALSE,
  colors = NULL,
  transparent = FALSE,
  res = 150,
  keep_ly = FALSE,
  simplify = TRUE
)
```

Arguments

music	a music object.
file	character, output file ending in .pdf or .png.
clef	character, include a music staff with the given clef. NA to suppress. See track() for details.
tab	logical, include tablature staff. NA to suppress. See track().
tuning	character, string tuning, only applies to tablature. See track().
string_names	label strings at beginning of tab staff. NULL (default) for non-standard tunings only, TRUE or FALSE for force on or off completely.
header	a named list of arguments passed to the header of the LilyPond file. See lilypond() details.
paper	a named list of arguments for the LilyPond file page layout. See lilypond() details.
midi	logical, also output an corresponding MIDI file.
colors	a named list of LilyPond element color global overrides. See lilypond() for details.
transparent	logical, transparent background, png only.
res	numeric, resolution, png only. transparent = TRUE may fail when res exceeds ~150.
keep_ly	logical, keep the intermediary LilyPond file.
simplify	logical, uses simplify_phrase() to convert to simpler, more efficient LilyPond syntax.

Details

These functions allow you to render short, simple snippets of sheet music directly from a music object. This is useful when you do not need to build up from phrases to tracks to a full score. They treat music objects as a single voice for a single track. This simplifies the possible output but is very convenient when this is all you need.

These functions abstract the following pipeline,
`music |> phrase() |> track() |> score() |> render_*()`

for this simple edge case and directly expose the most relevant arguments.

All header list elements are character strings. The options for header include the following.

- title
- subtitle
- composer
- album
- arranger
- instrument
- meter
- opus
- piece
- poet
- copyright
- tagline

All paper list elements are numeric except page_numbers and print_first_page_number, which are logical. page_numbers = FALSE suppresses all page numbering. When page_numbers = TRUE, you can set print_first_page_number = FALSE to suppress printing of only the first page number. first_page_number is the number of the first page, defaulting to 1, and determines all subsequent page numbers. These arguments correspond to LilyPond paper block variables.

The options for paper include the following and have the following default values if not provided.

- textheight = 220
- linewidth = 150
- indent = 0
- fontsize = 20
- page_numbers = FALSE
- print_first_page_number = TRUE
- first_page_number = 1

textheight = 150 is the default, but for music snippet rendering, a value must be provided explicitly via paper when rendering to png. Otherwise for png outputs the height is cropped automatically rather than remaining a full page. See lilypond() for details.

Passing arguments to header can completely or partially prevent cropping in both directions, which must then be done manually with linewidth and textheight. This is all based on underlying LilyPond behavior.

If music contains lyrics and there are rests in the note sequence, note-lyric alignment is maintained automatically when these functions remove the lyric timesteps corresponding to the rests prior to sending to LilyPond. LilyPond skips rests when engraving lyrics and expects a shortened lyrics sequence in comparison to how tabr matches by timestep including rests. This is in contrast to track(), for which you have to shorten the lyrics object yourself prior to combining with a phrase object that has rests.

Value

nothing returned; a file is written.

See Also

plot_music(), phrase(), track(), score(), lilypond(), tab()

Examples

```
x <- "a,4;5*5 b,- c cgc'e'~ cgc'e'1 e'4;2 c';3 g;4 c;5 ce'1;51"
x <- as_music(x)

y <- "a,,4;3*5 b,,- c, c,g,c~ c,g,c1 c4;1 g,;2 c,;3 g,;2 c,c1;31"
y <- as_music(y)

z <- as_music("a,4 b, r c~ c2 d", lyrics = as_lyrics("A2 B2 . C3 . D3"))

## Not run:
if(tabr_options()$lilypond != ""){ # requires LilyPond installation
  outfile <- file.path(tempdir(), "out.pdf")
  render_music(x, outfile)

  outfile <- file.path(tempdir(), "out.png")
  render_music(x, outfile, "treble_8", tab = TRUE)

  render_music_tc(x, outfile)
  render_music_bc(x, outfile)

  render_music_tab(x, outfile)
  render_music_guitar(x, outfile)
  render_music_bass(y, outfile)

  # lyrics example
  render_music_guitar(z, outfile)
}

## End(Not run)
```

---

repeats *Repeat phrases*

Description

Create a repeat section in LilyPond readable format.

Usage

```
rp(phrase, n = 1)

pct(phrase, n = 1, counter = FALSE, step = 1, reset = TRUE)

volta(phrase, n = 1, endings = NULL, silent = FALSE)
```

Arguments

phrase	a phrase object or equivalent string to be repeated.
n	integer, number of repeats of phrase (one less than the total number of plays).
counter	logical, if TRUE, print the percent repeat counter above the staff, applies only to *measure* repeats of more than two repeats (n > 2).
step	integer, print the *measure* percent repeat counter above the staff only at every step measures when counter = TRUE.
reset	logical, percent repeat counter and step settings are only applied to the single pct() call and are reset afterward. If reset = FALSE, the settings are left open to apply to any subsequent percent repeat sections in a track.
endings	a single phrase or a list of phrases, alternate endings.
silent	if TRUE, no text will be printed above the staff at the beginning of a volta section. See details.

Details

These functions wraps a phrase object or a character string in LilyPond repeat syntax. The most basic is rp() for basic wrapping a LilyPond unfold repeat tag around a phrase. This repeats the phrase n times, but it is displayed in the engraved sheet music fully written out as a literal propagation of the phrase with no repeat notation used to reduce redundant presentation. The next is pct(), which wraps a percent() repeat tag around a phrase. This is displayed in sheet music as percent repeat notation whose specific notation changes based on the length of the repeated section of music, used for beats or whole measures. volta() wraps a phrase in a volta() repeat tag, used for long repeats of one or more full measures or bars of music, optionally with alternate endings.

Note that basic strings should still be interpretable as a valid musical phrase by LilyPond and such strings will be coerced to the phrase class by these functions. For example, a one-measure rest, "r1", does not need to be a phrase object to work with these functions, nor does any other character string explicitly written out in valid LilyPond syntax. As always, see the LilyPond documentation if you are not familiar with LilyPond syntax.

VOLTA REPEAT: When silent = TRUE there is no indication of the number of plays above the staff at the start of the volta section. This otherwise happens automatically when the number of repeats is greater than one and no alternate endings are included (which are already numbered). This override creates ambiguity on its own, but is important to use multiple staves are present and another staff already displays the text regarding the number or plays. This prevents printing the same text above every staff.

PERCENT REPEAT: As indicated in the parameter descriptions, the arguments counter and step only apply to full measures or bars of music. It does not apply to shorter beats that are repeated using pct().

Value

a phrase.

See Also

phrase()

Examples

```
x <- phrase("c ec'g' ec'g'", "4 4 2", "5 432 432")
e1 <- phrase("a", 1, 5) # ending 1
e2 <- phrase("b", 1, 5) # ending 2

rp(x) # simple unfolded repeat, one repeat or two plays
rp(x, 3) # three repeats or four plays

pct(x) # one repeat or two plays
pct(x, 9, TRUE, 5) # 10 plays, add counter every 5 steps
pct(x, 9, TRUE, 5, FALSE) # as above, but do not reset counter settings

volta(x) # one repeat or two plays
volta(x, 1, list(e1, e2)) # one repeat with alternate ending
volta(x, 4, list(e1, e2)) # multiple repeats with only one alternate ending
volta(x, 4) # no alternates, more than one repeat
```

---

rest *Create rests*

Description

Create multiple rests efficiently with a simple wrapper around `rep()` using the `times` argument.

Usage

```
rest(x, n = 1)
```

Arguments

x	integer, duration.
n	integer, number of repetitions.

Value

a character string.

Examples

```
rest(c(1, 8), c(1, 4))
```

scale-deg *Scale degrees and mappings*

Description

These functions assist with mapping between scale degrees, notes and chords.

Usage

```
scale_degree(
  notes,
  key = "c",
  scale = "diatonic",
  use_root = TRUE,
  strict_accidentals = TRUE,
  naturalize = FALSE,
  roman = FALSE
)

scale_note(deg, key = "c", scale = "diatonic", collapse = FALSE, ...)

note_in_scale(
  notes,
  key = "c",
  scale = "diatonic",
  use_root = TRUE,
  strict_accidentals = TRUE
)

chord_degree(
  notes,
  key = "c",
  scale = "diatonic",
  strict_accidentals = TRUE,
  naturalize = FALSE,
  roman = FALSE
)

is_in_scale(notes, key = "c", scale = "diatonic", strict_accidentals = TRUE)
```

Arguments

notes	character, a string of notes.
key	character, key signature (or root note) for scale, depending on the type of scale.
scale	character, the suffix of a supported scale_* function.
use_root	logical, use lowest pitch in chord. Otherwise yield an NA in output.

strict_accidentals	logical, whether representation must match key and scale. See details.
naturalize	logical, whether to naturalize any sharps or flats before obtaiuning the scale degree.
roman	logical, return integer scale degrees as Roman numerals.
deg	integer, roman class, or character roman, the scale degree.
collapse	logical, collapse result into a single string ready for phrase construction.
...	additional arguments passed to the scale function, e.g., sharp = FALSE for scale_chromatic().

Details

Obtain the scale degree of a note in a supported scale with scale_degree(). This function works on any noteworthy string. It ignores octave numbering. Rests and any note not explicitly in the scale return NA. If deg is greater than the number of degrees in the scale, it is recycled, e.g., in C major 8 starts over as C.

By default, flats and sharps checked strictly against the scale. Setting strict_accidentals = FALSE will convert any flats or sharps present, if necessary based on the combination of key signature and scale. The chromatic scale is a special case where strict accidental is always ignored.

Not any arbitrary combination of valid key and valid scale is valid. For example, key = "am" and scale = "harmonic" is valid, but not with key = "a".

note_in_scale() is a wrapper around scale_degree(). To check if full chords are diatonic to the scale, see is_diatonic().

The inverse of scale_degree() is scale_note(), for obtaining the note associated with a scale degree. This could be done simply by calling a scale_* function and indexing its output directly, but this wrapper is provided to complement scale_degree(). Additionally, it accepts the common Roman numeral input for the degree. This can be with the roman class or as a character string. Degrees return NA if outside the scale degree range.

Value

integer, or roman class if roman = TRUE for scale_degree(); character for scale_note().

See Also

scale-helpers(), is_diatonic()

Examples

```
scale_degree("r c, e3 g~ g s g# ceg")
note_in_scale("r c, e3 g~ g s g# ceg")

scale_degree("c e g", roman = TRUE)

scale_degree("c c# d_ e", key = "d")
scale_degree("c c# d_ e", key = "d", strict_accidentals = FALSE)

scale_degree("c, e_3 g' f#ac#", use_root = FALSE)
scale_degree("c, e_3 g' f#ac#", naturalize = TRUE) # lowest chord pitch: c#
```

```
scale_degree("c# d_ e_' e4 f f# g", key = "c#", scale = "chromatic")

scale_note(1:3, key = "am")
scale_note(c(1, 3, 8), "d", collapse = TRUE)
all(sapply(list(4, "IV", as.roman(4)), scale_note) == "f")

x <- "d dfa df#a f#ac#"
chord_degree(x, "d")
is_in_scale(x, "d")
```

---

scale-helpers *Scale helpers*

Description

Helper functions for working with musical scales.

Usage

```
scale_diatonic(key = "c", collapse = FALSE, ignore_octave = FALSE)

scale_major(key = "c", collapse = FALSE, ignore_octave = FALSE)

scale_minor(key = "am", collapse = FALSE, ignore_octave = FALSE)

scale_harmonic_minor(key = "am", collapse = FALSE, ignore_octave = FALSE)

scale_hungarian_minor(key = "am", collapse = FALSE, ignore_octave = FALSE)

scale_melodic_minor(
  key = "am",
  descending = FALSE,
  collapse = FALSE,
  ignore_octave = FALSE
)

scale_jazz_minor(key = "am", collapse = FALSE, ignore_octave = FALSE)

scale_chromatic(
  root = "c",
  collapse = FALSE,
  sharp = TRUE,
  ignore_octave = FALSE
)
```

Arguments

key	character, key signature.
collapse	logical, collapse result into a single string ready for phrase construction.
ignore_octave	logical, strip octave numbering from scales not rooted on C.
descending	logical, return the descending scale, available as a built-in argument for the melodic minor scale, which is different in each direction.
root	character, root note.
sharp	logical, accidentals in arbitrary scale output should be sharp rather than flat.

Details

For valid key signatures, see keys().

Value

character

See Also

keys(), mode-helpers()

Examples

```
scale_diatonic(key = "dm")
scale_minor(key = "dm")
scale_major(key = "d")

scale_chromatic(root = "a")

scale_harmonic_minor("am")
scale_hungarian_minor("am")

identical(scale_melodic_minor("am"), scale_jazz_minor("am"))
rev(scale_melodic_minor("am", descending = TRUE))
scale_jazz_minor("am")
```

---

scale_chords *Diatonic chords*

Description

Obtain an ordered sequence of the diatonic chords for a given scale, as triads or sevenths.

Usage

```
scale_chords(
  root = "c",
  scale = "major",
  type = c("triad", "seventh"),
  collapse = FALSE
)
```

Arguments

root	character, root note or starting position of scale.
scale	character, a valid named scale, referring to one of the existing scale_* functions.
type	character, type of chord, triad or seventh.
collapse	logical, collapse result into a single string ready for phrase construction.

Value

character

Examples

```
scale_chords("c", "major")
scale_chords("a", "minor")
scale_chords("a", "harmonic minor")
scale_chords("a", "melodic minor")
scale_chords("a", "jazz minor")
scale_chords("a", "hungarian minor")

scale_chords("c", "major", "seventh", collapse = TRUE)
scale_chords("a", "minor", "seventh", collapse = TRUE)
```

---

score *Create a music score*

Description

Create a music score from a collection of tracks.

Usage

```
score(track, chords = NULL, chord_seq = NULL)
```

Arguments

track	a track table consisting of one or more tracks.
chords	an optional named list of chords and respective fingerings generated by chord_set, for inclusion of a top center chord diagram chart.
chord_seq	an optional named vector of chords and their durations, for placing chord diagrams above staves in time.

Details

Score takes track tables generated by track() and fortifies them as a music score. It optionally binds tracks with a set of chord diagrams. There may be only one track in track() as well as no chord information passed, but for consistency score() is still required to fortify the single track as a score object that can be rendered by tab().

Value

a tibble data frame

See Also

phrase(), track(), trackbind()

Examples

```
x <- phrase("c ec'g' ec'g'", "4 4 2", "5 432 432")
x <- track(x)
score(x)
```

---

sf_phrase *Create a musical phrase from string/fret combinations*

Description

Create a musical phrase from character strings that define string numbers, fret numbers and note metadata. This function is a wrapper around phrase().

Usage

```
sf_phrase(
  string,
  fret = NULL,
  info = NULL,
  key = "c",
  tuning = "standard",
  to_notes = FALSE,
  bar = NULL
)
```

```
sfp(
  string,
  fret = NULL,
  info = NULL,
  key = "c",
  tuning = "standard",
  to_notes = FALSE,
  bar = NULL
)

sf_note(...)

sfn(...)
```

Arguments

string	character, space-delimited or vector. String numbers associated with notes. Alternatively, provide all information here in a single space-delimited string and ignore fret and info. See details.
fret	character, space-delimited or vector (or integer vector) of fret numbers associated with notes. Same number of timesteps as string.
info	character, space-delimited or vector (or integer vector if simple durations) giving metadata associated with notes. Same number of timesteps as string.
key	character, key signature or just specify "sharp" or "flat".
tuning	character, instrument tuning.
to_notes	logical, return only the mapped notes character string rather than the entire phrase object.
bar	character or NULL (default). Terminates the phrase with a bar or bar check. See details for phrase(). Also see the LilyPond help documentation on bar notation for all the valid options.
...	arguments passed to sf_phrase().

Details

Note: This alternate specification wrapper is not receiving further support and will be removed in a future version of tabr.

This alternate syntax allows for specifying string/fret combinations instead of unambiguous pitch as is used by phrase(). In order to remove ambiguity, it is critical to specify the instrument string tuning and key signature. It essentially uses string and fret in combination with a known tuning and key signature to generate notes for phrase(). info is passed straight through to phrase(), as is string once it is done being used to help derive notes.

See the main function phrase for general details on phrase construction.

Value

a phrase.

Comparison with `phrase()`

This function is a wrapper function for users not working with musical notes (what to play), but rather just position on the guitar neck (where to play). This approach has conveniences, but is more limiting. In order to remove ambiguity, it is necessary to specify the instrument `tuning` and the `key` signature.

In the standard approach with `phrase()` you specify what to play; specifying exactly where to play is optional, but highly recommended (by providing `string`). With `sf_phrase()`, the `string` argument is of course required along with `fret`. But any time the tuning changes, this "where to play" method breaks down and must be redone. It is more robust to provide the string and pitch rather than the string and fret. The `key` is additionally important because it is the only way to indicate if accidentals should be notated as sharps or flats.

This wrapper also increases redundancy and typing. In order to specify rests `r`, silent rests `s`, and tied notes `~`, these must now be providing in parallel in both the `string` and `fret` arguments, whereas in the standard method using `phrase()`, they need only be provided once to `notes`. A mismatch will throw an error. Despite the redundancy, this is helpful for ensuring proper match up between `string` and `fret`, which is essentially a dual entry method that aims to reduce itself inside `sf_phrase()` to a single `notes` string that is passed internally to `phrase()`.

The important thing to keep in mind is that by its nature, this method of writing out music does not lend itself well to high detail. Tabs that are informed by nothing but string and fret number remove a lot of important information, and those that attempt to compensate with additional symbols in say, an ascii tab, are difficult to read. This wrapper function providing this alternative input method to `phrase()` does its job of allowing users to create phrase objects that are equivalent to standard `phrase()`-generated objects, including rests and ties. But practice and comfort working with `phrase()` is is highly recommended for greater control of development support.

The function `sfp()` is a convenient shorthand wrapper for `sf_phrase()`. `sf_note()` and the alias `sfn()` are wrappers around `sf_phrase()` that force `to_notes = TRUE`.

Single-string input

Another way to use `sf_phrase()` is to provide all musical input to `string` and ignore `fret` and `info` as explicit arguments. Providing all three explicit arguments more closely mimics the inputs of `phrase()` and is useful when you have this information as three independent sources. However, in some cases the single-argument input method can reduce typing, though this depends on the phrase. More importantly, it allow you to reason about your musical inputs by time step rather than by argument. If you provide all three components as a single character string to the `string` argument, leaving both `fret` and `info` as `NULL`, then `sf_phrase()` will decompose `string` into its three component parts internally.

There are some rules for single-argument input. The three components are separated by semicolons as `"string;fret;info"`. For example, `"3;7x7;4"` means begin on the third string (infer higher number strings muted). The frets are 7th and 7th, meaning two notes are played. When an `x` is present in the second entry it means a string is not played. This is how it is inferred that the string numbers starting from the third string are strings 3 and 1 rather than 3 and 2 in this example. The 4 indicates a quarter note since it is part of the third entry where the additional `info` is specified. This is contextual. For example, an `x` here would still indicate a dead note, rather than an unplayed string in the second entry, so this is contextual.

A bonus when using this input method is that explicit `string` and `info` values persist from one timestep to the next. Neither needs to be provided again until there is a change in value. For

example, `"3;7x7;4 7x7 ;7x7;1"` repeats the string and info values from timestep one for timestep two. In timestep three, string numbers repeat again, but the duration changes from quarter note to whole note.

Note that except when both `string` and `info` are repeating and only fret numbers are provided (see timestep two above), two semicolons must be present so that it is unambiguous whether the sole missing component is a `string` or `info` (see timestep three).

Ambiguity would arise from a case like `"4;4"` without the second semicolon. This type of indexing was chosen over using two different delimiters.

If a rest, `r` or `s`, is provided for the `fret` entry, then the `string` entry is ignored. When using this input method, ties `~` are given in the info entry.

See the examples for a comparison of two identical phrases specified using both input methods for `sf_phrase()`.

See Also

`phrase()`

Examples

```
sf_phrase("5 4 3 2 1", "1 3 3 3 1", "8*4 1", key = "b_")
sf_phrase("6 6 12 1 21", "133211 355333 11 (13) (13)(13)", "4 4 8 8 4",
          key = "f")
sfp("6*2 1*4", "000232*2 2*4", "4 4 8*4", tuning = "dropD", key = "d")

# compare with single-argument input
s <- "3*5 53~*3 543*2 643"
f <- "987*2 775 553 335 77~*3 545 325 210"
i <- "2*3 4. 16 4.*3 4*3"
p1 <- sfp(s, f, i)

# Nominally shorter syntax, but potentially much easier to reason about
p2 <- sfp("3;987;2*2 775 ;553;4. ;335;16 5;7x7;4.~*3 ;545;4 325 6;2x10;")

identical(p1, p2)
```

---

simplify_phrase *Simplify the LilyPond syntax of a phrase*

Description

This function can be used to simplify the LilyPond syntax of a phrase. Not intended for direct use. See details.

Usage

```
simplify_phrase(phrase)
```

Arguments

phrase	a phrase object.

Details

This function not intended to be used directly, but is available so that you can see how LilyPond syntax for phrases will be transformed by default in the process of creating a LilyPond file. This function is used by the `lilypond()` function and associated `render_*` functions. When using `lilypond()` directly, this can be controlled by the `simplify` argument.

The result of this function is a character string containing simpler, more efficient LilyPond syntax. It can be coerced back to a phrase with `as_phrase()`, but its print method colors will no longer display properly. More importantly, this simplification removes any possibility of transforming the phrase back to its original inputs. The more complex but nicely structured original representation does a better job at maintaining reasonable possibility of one to one transformation between a phrase object and the inputs that it was built from.

Value

character

Examples

```
notes <- "a~ a b c' c'e'g'~ c'e'g'"
info <- "8.. 8..-. 8- 8-^ 4.^ 4."
(x <- p(notes, info))
as_phrase(simplify_phrase(x))

(x <- p(notes, info, 5))
as_phrase(simplify_phrase(x))
```

---

single-bracket *Single bracket methods for tabr classes*

Description

Single bracket indexing and assignment. See `tabr-methods()` for more details on methods for tabr classes.

Usage

```
## S3 method for class 'noteworthy'
x[i]

## S3 method for class 'noteinfo'
x[i]

## S3 method for class 'music'
```

```
x[i]

## S3 method for class 'lyrics'
x[i]

## S3 replacement method for class 'noteworthy'
x[i] <- value

## S3 replacement method for class 'noteinfo'
x[i] <- value

## S3 replacement method for class 'music'
x[i] <- value

## S3 replacement method for class 'lyrics'
x[i] <- value
```

Arguments

x	object.
i	index.
value	values to assign at index.

See Also

tabr-methods(), note-metadata()

Examples

```
# noteworthy class examples
x <- as_noteworthy("a, b, c ce_g d4f#4a4")
x[3:4]
x[-2]
x[2] <- paste0(transpose(x[2], 1), "~")
x

# noteinfo class examples
x <- as_noteinfo(c("4-", "t8(", "t8)", "t8x", "8^", "16"))
x[2:4]
x[-1]
x[5:6] <- c("16^", "8")
x
x[x == "4-"]

# music class examples
x <- as_music("c,~4 c,1 c'e_'g'4-.*4")
x[1:3]
x[-c(1:2)]
x[3:6] <- "c'e'g'8"
x
```

`string_unfold` *Fold and unfold strings*

Description

Fold or unfold a string on the expansion operator.

Usage

```
string_unfold(x)

string_fold(x, n = 3)
```

Arguments

`x`	character string, should be valid notes or note info such as beats.
`n`	integer, minimum number of consecutive repeated values to warrant folding, defaults to 3.

Details

These function work on arbitrary stings. They do not perform a noteworthy check. This allows them to work for `info` strings as well. Make sure your strings are properly formatted. `string_fold()` always collapses the output string as space-delimited.

Value

character

Examples

```
time <- "8*3 16 4.. 16 16 2 2 4. 8 4 4 8*4 1"
x <- string_unfold(time)
x
string_fold(x) == time

notes <- "a, b, c d e f g# a r ac'e' a c' e' c' r r r a"
x <- string_fold(notes)
x
string_unfold(x) == notes
```

tab *Render sheet music with LilyPond*

Description

Render sheet music/tablature from a music score with LilyPond.

Usage

```
tab(
  score,
  file,
  key = "c",
  time = "4/4",
  tempo = "2 = 60",
  header = NULL,
  paper = NULL,
  string_names = NULL,
  endbar = "|.",
  midi = TRUE,
  colors = NULL,
  crop_png = TRUE,
  transparent = FALSE,
  res = 150,
  keep_ly = FALSE,
  simplify = TRUE,
  details = FALSE
)

render_tab(
  score,
  file,
  key = "c",
  time = "4/4",
  tempo = "2 = 60",
  header = NULL,
  paper = NULL,
  string_names = NULL,
  endbar = "|.",
  midi = TRUE,
  colors = NULL,
  crop_png = TRUE,
  transparent = FALSE,
  res = 150,
  keep_ly = FALSE,
  simplify = TRUE,
  details = FALSE
```

```
)

render_score(
  score,
  file,
  key = "c",
  time = "4/4",
  tempo = "2 = 60",
  header = NULL,
  paper = NULL,
  endbar = "|.",
  colors = NULL,
  crop_png = TRUE,
  transparent = FALSE,
  res = 150,
  keep_ly = FALSE,
  simplify = TRUE,
  details = FALSE
)

render_midi(score, file, key = "c", time = "4/4", tempo = "2 = 60")
```

Arguments

score	a score object.
file	character, output file ending in .pdf or .png for sheet music or tablature for `score()`. May include an absolute or relative path. For `render_midi()`, a file ending in `.mid`.
key	character, key signature, e.g., `c`, `b_`, `f#m`, etc.
time	character, defaults to `"4/4"`.
tempo	character, defaults to `"2 = 60"`. Set to `NULL` to suppress display of the time signature in the output.
header	a named list of arguments passed to the header of the LilyPond file. See `lilypond()` for details.
paper	a named list of arguments for the LilyPond file page layout. See `lilypond()` for details.
string_names	label strings at beginning of tab staff. `NULL` (default) for non-standard tunings only, `TRUE` or `FALSE` for force on or off completely.
endbar	character, the global end bar.
midi	logical, output midi file in addition to sheet music.
colors	a named list of LilyPond element color overrides. See `lilypond()` for details.
crop_png	logical, see `lilypond()` for details.
transparent	logical, transparent background, png only.
res	numeric, resolution, png only. `transparent = TRUE` may fail when `res` exceeds ~150.

keep_ly	logical, keep the intermediary LilyPond file.
simplify	logical, uses simplify_phrase() to convert to simpler, more efficient LilyPond syntax for the LilyPond file before rendering it.
details	logical, set to TRUE to print LilyPond log output to console. Windows only.

Details

Generate a pdf or png of a music score using the LilyPond music engraving program. Output format is inferred from file extension. This function is a wrapper around lilypond(), the function that creates the LilyPond (.ly) file.

render_score() renders score() to pdf or png. render_midi() renders a MIDI file based on score(). This is still done via LilyPond. The sheet music is created automatically in the process behind the scenes but is deleted and only the MIDI output is retained.

tab() or render_tab() (equivalent) produces both the sheet music and the MIDI file output by default and includes other arguments such as the tablature-relevant argument string_names. This is the all-purpose function. Also use this when you intend to create both a sheet music document and a MIDI file.

Remember that whether a track contains a tablature staff, standard music staff, or both, is defined in each individual track object contained in score(). It is the contents you have assembled inscore() that dictate what render function you should use. render_tab() is general and always works, but render_score() would not be the best choice when a tablature staff is present unless you accept the default string naming convention.

render_midi() is different from midily() and miditab(), whose purpose is to create sheet music from an existing MIDI file using a LilyPond command line utility.

For Windows users, add the path to the LilyPond executable to the system path variable. For example, if the file is at C:/lilypond-2.24.2/bin/lilypond.exe, then add C:/lilypond-2.24.2/bin to the system path.

Value

nothing returned; a file is written.

See Also

lilypond(), render_chordchart(), miditab()

Examples

```
if(tabr_options()$lilypond != ""){
  x <- phrase("c ec'g' ec'g'", "4 4 2", "5 432 432")
  x <- track(x)
  x <- score(x)
  outfile <- file.path(tempdir(), "out.pdf")
  tab(x, outfile) # requires LilyPond installation
}
```

tabr *tabr: Music notation syntax, manipulation, analysis and transcription in R.*

Description

The tabr package provides a music notation syntax and a collection of music programming functions for generating, manipulating, organizing and analyzing musical information in R. The music notation framework facilitates creating and analyzing music data in notation form.

Details

Music syntax can be entered directly in character strings, for example to quickly transcribe short pieces of music. The package contains functions for directly performing various mathematical, logical and organizational operations and musical transformations on special object classes that facilitate working with music data and notation. The same music data can be organized in tidy data frames for a familiar and powerful approach to the analysis of large amounts of structured music data. Functions are available for mapping seamlessly between these formats and their representations of musical information.

The package also provides an API to 'LilyPond' (https://lilypond.org/) for transcribing musical representations in R into tablature ("tabs") and sheet music. 'LilyPond' is open source music engraving software for generating high quality sheet music based on markup syntax. The package generates 'LilyPond' files from R code and can pass them to the 'LilyPond' command line interface to be rendered into sheet music PDF files or inserted into R markdown documents.

The package offers nominal MIDI file output support in conjunction with rendering sheet music. The package can read MIDI files and attempts to structure the MIDI data to integrate as best as possible with the data structures and functionality found throughout the package.

tabr offers a useful but limited LilyPond API and is not intended to access all LilyPond functionality from R, nor is transcription via the API the entire scope of tabr. If you are only creating sheet music on a case by case basis, write your own LilyPond files manually. There is no need to use tabr or limit yourself to its existing LilyPond API. If you are generating music notation programmatically, tabr provides the ability to do so in R and has the added benefit of converting what you write in R code to the LilyPond file format to be rendered as printable guitar tablature.

While LilyPond is listed as a system requirement for tabr, you can use the package for music analysis without installing LilyPond if you do not intend to render tabs.

tabr-c *Concatenate for tabr classes*

Description

Several methods are implemented for the classes noteworthy, noteinfo, and music. See tabr-methods() for more details on methods for tabr classes.

Usage

```
## S3 method for class 'noteworthy'
c(...)

## S3 method for class 'noteinfo'
c(...)

## S3 method for class 'music'
c(...)

## S3 method for class 'lyrics'
c(...)

## S3 method for class 'phrase'
c(...)
```

Arguments

...	objects.

See Also

tabr-methods(), note-metadata()

Examples

```
# noteworthy class examples
x <- "a b c"
c(x, x)
c(as_noteworthy(x), x)

# noteinfo class examples
x <- "4- t8( t8)( t8) 4*2"
c(as_noteinfo(x), x)

# music class examples
x <- "c,~4 c,1 c'e_'g'4-.*2"
c(as_music(x), x)

# phrase class examples
c(phrase(x), x)
```

---

tabr-head *Head and tail for tabr classes*

Description

Several methods are implemented for the classes noteworthy, noteinfo, and music. See tabr-methods() for more details on methods for tabr classes.

Usage

```
## S3 method for class 'noteworthy'
head(x, ...)

## S3 method for class 'noteinfo'
head(x, ...)

## S3 method for class 'music'
head(x, ...)

## S3 method for class 'lyrics'
head(x, ...)

## S3 method for class 'noteworthy'
tail(x, ...)

## S3 method for class 'noteinfo'
tail(x, ...)

## S3 method for class 'music'
tail(x, ...)

## S3 method for class 'lyrics'
tail(x, ...)
```

Arguments

x	object.
...	number of elements to return.

See Also

tabr-methods(), note-metadata()

Examples

```
# noteworthy class examples
x <- "a b c d e f g"
head(x, 2)
head(as_noteworthy(x), 2)
tail(as_noteworthy(x), 2)

# noteinfo class examples
x <- "4x 4-.*8 2 4"
head(as_noteinfo(x))
tail(as_noteinfo(x))

# music class examples
x <- "c,~4 c,1 c'e_'g'4-."
```

```
head(as_music(x), 2)
tail(as_music(x), 2)
```

---

tabr-length *Length for tabr classes*

Description

Several methods are implemented for the classes `noteworthy`, `noteinfo`, and `music`. See `tabr-methods()` for more details on methods for tabr classes.

Usage

```
## S3 method for class 'noteworthy'
length(x)

## S3 method for class 'noteinfo'
length(x)

## S3 method for class 'music'
length(x)

## S3 method for class 'lyrics'
length(x)
```

Arguments

x	object.

See Also

`tabr-methods()`, `note-metadata()`

Examples

```
# noteworthy class examples
x <- "a b c"
length(x)
length(as_noteworthy(x))
length(as_noteworthy("a b*2 c*2"))

# noteinfo class examples
x <- "4- t8( t8)( t8) 4*2"
length(x)
length(as_noteinfo(x))

# music class examples
x <- "c,~4 c,1 c'e_'g'4-.*4"
length(x)
length(as_music(x))
```

tabr-methods *Summary of implemented S3 generic methods*

Description

Several methods are implemented for the classes noteworthy, noteinfo, music and lyrics. See further below for details on limited implementations for the phrase class.

Arguments

x	object.
i	index.
value	values to assign at index.
...	additional arguments.

Details

In addition to custom print and summary methods, the following methods have been implemented for all four classes: [, [<-, [[, [[<-, length(), c(), rep(), rev(), head() and tail(). Logical operators are also implemented for noteworthy strings.

Methods length() and c()

The implementation of length() is equivalent to n_steps(). They access the same attribute, returning the number of timesteps in the object. This gives the same result even when the underlying string is in space-delimited format. To obtain the character string length, coerce with as.character() or any other function that would have the same effect.

The implementation of c() for these classes is strict and favors the object class in question. This is different from how c() might normally behave, coercing objects of different types such as numeric and character to character.

For these four classes, c() is strict in that it will return an error if attempting to concatenate one of these classes with any other class besides character. This includes each other. While it would be possible to coerce a music object down to a noteworthy object or a noteinfo object, this is the opposite of the aggressive coercion these classes are intended to have with c() so this is not done.

While other classes such as numeric immediately return an error, any concatenation with character strings attempts to coerce each character string present to the given class. If coercion fails for any character class object, the usual error is returned concerning invalid notes or note info present. If coercion succeeds for all character strings, the result of c() is to concatenate the timesteps of all objects passed to it. The output is a new noteworthy, noteinfo ormusic object.

Methods rep() rev() head() and tail()

The rep() function is similar to c() except that it never has to consider other classes. You could pass a vector of objects to rep(), but doing so with c() will already have resolved all objects to the single class. Again, what matters is not the underlying length or elements in the character vector

the class is built upon, but the timesteps. `rep()` will extend `x` in terms of timesteps. You can also provide the `each` or `times` arguments.

`rev()`, `head()` and `tail()` work similarly, based on the sequence of timesteps, not the character vector length.

Remember that this accounts not only for vectors of length one that contain multiple timesteps in space-delimited time format, but also that multiple timesteps can be condensed even in space-delimited time format with the `*` expansion operator. For example, `"a'*4 b'*2"` has six timesteps in this form as well as in vector form. The object length is neither one nor two. All of these generic method implementations work in this manner.

Square brackets

Single and double bracket subsetting by index work similarly to what occurs with lists. Single bracket subsetting returns the same object, but only containing the indexed timesteps. Double bracket subsetting only operates on a single timestep and extracts the character string value.

For assignment, single and double brackets change the value at timesteps and return the same object, but again double brackets only allow indexing a single timestep. Double bracket indexing is mostly useful for combining the steps of extracting a single value and discarding the special class in one command.

Limited phrase implementations

Methods implemented for the `phrase` class are limited to `c()` and `rep()`. Due to the complex LilyPond syntax, applying most of the functions above directly to phrases is problematic. `c()` is implemented like it is for the other classes. `rep()` is restricted in that it can only repeat the entire phrase sequence, not the timesteps within. However, you can convert a phrase class back to `noteworthy` and `noteinfo` objects (under reasonable conditions). See `notify()`.

One exception made for `phrase` objects with respect to concatenation is that an attempt to concatenate any combination of phrase and music objects, in any order, results in coercion to a new phrase. This happens even in a case where the first object in the sequence is a music object (thus calling `c.music()` rather than `c.phrase()`). It will subsequently fall back to `c.phrase()` in that case.

See Also

`note-logic()`, `note-metadata()`

Examples

```
# noteworthy class examples
x <- as_noteworthy("a, b, c ce_g d4f#4a4")
x
x[3:4]
x[-2]
x[2] <- paste0(transpose(x[2], 1), "~")
x
length(x) # equal to number of timesteps
c(x, x)
tail(rep(x, times = c(1, 2, 1, 3, 1)))
```

```
# noteinfo class examples
x <- as_noteinfo(c("4-", "t8(", "t8)", "t8x", "8^", "16"))
x
x[2:4]
x[-1]
x[5:6] <- c("16^", "8")
x
x[x == "4-"]
c(x[1], x[2]) == c(x[1:2])
head(rep(x, each = 2))

# music class examples
x <- as_music("c,~4 c,1 c'e_'g'4-.*4")
x
x[1:3]
x[-c(1:2)]
x[3:6] <- "c'e'g'8"
x
c(x[1], x[1]) == x[c(1, 1)]
rev(x)

x[[3]]
x[[3]] <- "b_t8"
x
```

---

tabr-rep *Repeat for tabr classes*

Description

Several methods are implemented for the classes `noteworthy`, `noteinfo`, and `music`. See `tabr-methods()` for more details on methods for tabr classes.

Usage

```
## S3 method for class 'noteworthy'
rep(x, ...)

## S3 method for class 'noteinfo'
rep(x, ...)

## S3 method for class 'music'
rep(x, ...)

## S3 method for class 'lyrics'
rep(x, ...)

## S3 method for class 'phrase'
rep(x, ...)
```

Arguments

x	object.
...	additional arguments. Not accepted for phrase objects.

See Also

tabr-methods(), note-metadata()

Examples

```
# noteworthy class examples
x <- "a b c"
rep(x, 2)
rep(as_noteworthy(x), 2)

# noteinfo class examples
x <- "4x 4-.*2 2"
rep(as_noteinfo(x), times = c(2, 1, 1, 2))

# music class examples
x <- "c,~4 c,1 c'e_'g'4-."
rep(as_music(x), each = 2)

# phrase class examples
rep(phrase(x), 2)
```

---

tabr-rev *Reverse for tabr classes*

Description

Several methods are implemented for the classes noteworthy, noteinfo, and music. See tabr-methods() for more details on methods for tabr classes.

Usage

```
## S3 method for class 'noteworthy'
rev(x)

## S3 method for class 'noteinfo'
rev(x)

## S3 method for class 'music'
rev(x)

## S3 method for class 'lyrics'
rev(x)
```

Arguments

x	object.

See Also

tabr-methods(), note-metadata()

Examples

```
# noteworthy class examples
x <- "a b c"
rev(x)
rev(as_noteworthy(x))

# noteinfo class examples
x <- "4x 4-.*2 2"
rev(as_noteinfo(x))

# music class examples
x <- "c,~4 c,1 c'e_'g'4-."
rev(as_music(x))
```

---

tabrSyntax *tabr syntax*

Description

A data frame containing descriptions of syntax used in phrase construction in tabr.

Usage

```
tabrSyntax
```

Format

A data frame with 3 columns for syntax description, operators and examples.

tabr_options *Options*

Description

Options for tabr package.

Usage

```
tabr_options(...)
```

Arguments

...	a list of options.

Details

Currently only `lilypond`, `midi2ly` and `python` are used. On Windows systems, if the system path for lilypond.exe, midi2ly and python.exe are not stored in the system PATH environmental variable, they must be provided by the user after loading the package.

Value

The function prints all set options if called with no arguments. When setting options, nothing is returned.

Examples

```
tabr_options()
lilypond_path <- "C:/lilypond-2.24.2/bin/lilypond.exe" # if installed here
tabr_options(lilypond = lilypond_path)
```

tie *Tied notes*

Description

Tie notes efficiently.

Usage

```
tie(x)

untie(x)
```

Arguments

x	character, a single chord.

Details

This function is useful for bar chords.

Value

a character string.

Examples

```
tie("e,b,egbe'")
```

---

to_tabr *Music notation syntax converters*

Description

Convert alternative representations of music notation to tabr syntax.

Usage

```
to_tabr(id, ...)

from_chorrrds(x, key = "c", guitar = FALSE, gc_args = list())

from_music21(x, accidentals = c("flat", "sharp"), output = c("music", "list"))
```

Arguments

id	character, suffix of from_* function, e.g., "chorrrds"
...	arguments passed to the function matched by id.
x	character, general syntax input. See details and examples for how inputs are structured for each converter.
key	key signature, used to enforce consistent use of flats or sharps.
guitar	logical, attempt to match input chords to known guitar chords in guitarChords(). Otherwise by default standard piano chords of consecutive pitches covering minimum pitch range are returned.
gc_args	named list of additional arguments passed to gc_info(), used when guitar = TRUE.
accidentals	character, represent accidentals, "flat" or "sharp".
output	character, type of output when multiple options are available.

Details

These functions convert music notation from other data sources into the style used by `tabr` for music analysis and sheet music transcription.

Value

noteworthy string for chorrrds; music string or list for music21.

Syntax converter for chorrrds

The input `x` is a character vector of chords output from the `chorrrds` package, as shown in the examples. Output is a noteworthy string object.

Some sources do not offer as complete or explicit information in order to make sheet music. However, what is available in those formats is converted to the extent possible and available function arguments can allow the user to add some additional specification. Different input syntax makes use of a different syntax converter. Depending on the format, different arguments may be available and/or required. The general wrapper function for all of the available syntax converters is `to_tabr()`. This function takes an `id` argument for the appropriate converter function. See examples.

For example, output from the `chorrrds` package that scrapes chord information from the Cifraclub website only provides chords, not note for note transcription data for any particular instrument. This means the result of syntax conversion still yields only chords, which is fine for data analysis but doesn't add anything useful for sheet music transcription.

The input in this case also does not specify distinct pitches by assigning octaves numbers to a chord's notes, not even the root note. It remains up to the user if they want to apply the information. By default, every chord starts in octave three. It is also ambiguous how the chord is played since all that is provided is a generic chord symbol. By default a standard chord is constructed if it can be determined.

Setting `guitar = TRUE` switches to using the `guitarChords()` dataset to find matching guitar chords using `gc_info()`, which can be provided additional arguments in a named list to `gc_args`. For guitar, this allows some additional control over the actual structure of the chord, its shape and position on the guitar neck. The options will never work perfectly for all chords in `chords`, but at a minimum, typical default component pitches will be determined and returned in `tabr` notation style.

Syntax converter for music21

The input `x` is a character vector of in music21 tiny notation syntax, as shown in the examples. Default output is a music object. Setting `output = "list"` returns a list of three elements: a noteworthy string, a note info string, and the time signature.

The recommendation for music21 syntax is to keep it simple. Do not use the letter `n` for explicit natural notes. Do not add text annotations such as lyrics. Double flats and sharps are not supported. The examples demonstrate what is currently supported.

Examples

```
# chorrrds package output
chords <- c("Bb", "Bbm", "Bbm7", "Bbm7(b5)", "Bb7(#5)/G", "Bb7(#5)/Ab")
```

```
from_chorrrds(chords)
to_tabr(id = "chorrrds", x = chords)

from_chorrrds(chords, guitar = TRUE)
to_tabr(id = "chorrrds", x = chords, guitar = TRUE)

# music21 tiny notation
x <- "4/4 CC#FF4.. trip{c#8eg# d'- e-' f g a'} D4~# D E F r B16"
from_music21(x)

from_music21(x, accidentals = "sharp")

from_music21(x, output = "list")
```

---

track *Create a music track*

Description

Create a music track from a collection of musical phrases.

Usage

```
track(
  phrase,
  clef = "treble_8",
  key = NA,
  tab = TRUE,
  tuning = "standard",
  voice = 1,
  lyrics = NA
)

track_guitar(
  phrase,
  clef = "treble_8",
  key = NA,
  tab = TRUE,
  tuning = "standard",
  voice = 1,
  lyrics = NA
)

track_tc(phrase, key = NA, voice = 1, lyrics = NA)

track_bc(phrase, key = NA, voice = 1, lyrics = NA)

track_bass(phrase, key = NA, voice = 1, lyrics = NA)
```

Arguments

phrase	a phrase object.
clef	character, include a music staff with the given clef. NA to suppress. See details.
key	character, key signature for music staff. See details.
tab	logical, include tablature staff. NA to suppress.
tuning	character, pitches describing the instrument string tuning or a predefined tuning ID. See tunings(). Defaults to standard guitar tuning; not relevant if tablature staff is suppressed.
voice	integer, ID indicating the unique voice phrase() belongs to within a single track (another track may share the same tab/music staff but have a different voice ID). Up to two voices are supported per track.
lyrics	a lyrics object or NA. See details.

Details

Musical phrases generated by phrase() are fortified in a track table. All tracks are stored as track tables, one per row, even if that table consists of a single track. track() creates a single-entry track table. See trackbind() for merging single tracks into a multi-track table. This is row binding that also properly preserves phrase and track classes.

There are various track_* functions offering sensible defaults based on the function suffix. The base track() function is equivalent to track_guitar(). See examples. Setting clef = NA or tab = NA suppresses the music staff or tablature staff, respectively. By default key = NA, in which case its inherits the global key from the key argument of various sheet music rendering functions. If planning to bind two tracks as one where they are given voice = 1 and voice = 2, respectively, they must also have a common key, even if key = NA.

lyrics should only be used for simple tracks that do not contain repeats. You also need to ensure the timesteps for lyrics align with those of phrase() in advance. Additionally, LilyPond does not engrave lyrics at rests or tied notes (excluding first note in tied sequence) so if Therefore, if phrase() contains rests and tied notes then the lyrics object should be subset to exclude these timesteps as well. This is in contrast to using render_music* functions, which handle this automatically for music objects.

Value

a tibble data frame

See Also

phrase(), score()

Examples

```
x <- phrase("c ec'g' ec'g'", "4 4 2", "5 4 4")
track(x) # same as track_guitar(x); 8va treble clef above tab staff
track_tc(x) # treble clef sheet music, no tab staff
track_bc(x) # bass clef sheet music, no tab staff
```

```
x <- phrase("c, g,c g,c", "4 4 2", "3 2 2")
track_bass(x) # includes tab staff and standard bass tuning
```

`trackbind` *Bind track tables*

Description

Bind together track tables by row.

Usage

```
trackbind(..., id)
```

Arguments

`...`	single-entry track data frames.
`id`	integer, ID vector indicating distinct tracks corresponding to distinct sheet music staves. See details.

Details

This function appends multiple track tables into a single track table for preparation of generating a multi-track score. `id` is used to separate staves in the sheet music/tablature output. A track's `voice` is used to separate distinct voices within a common music staff.

If not provided, `id` automatically propagates `1:n` for `n` tracks passed to `...` when binding these tracks together. This expresses the default assumption of one staff or music/tab staff pair per track. This is the typical use case.

Some tracks represent different voices that share the same staff. These should be assigned the same `id`, in which case you must provide the `id` argument. Up to two voices per track are supported. An error will be thrown if any two tracks have both the same `voice` and the same `id`. The pair must be unique. E.g., provide `id = c(1, 1)` when you have two tracks with `voice` equal to 1 and 2. See examples.

Note that the actual ID values assigned to each track do not matter; only the order in which tracks are bound, first to last.

Value

a tibble data frame

See Also

`phrase()`, `track()`, `score()`

Examples

```
x <- phrase("c ec'g' ec'g'", "4 4 2", "5 432 432")
x1 <- track(x)
x2 <- track(x, voice = 2)
trackbind(x1, x1)
trackbind(x1, x2, id = c(1, 1))
```

`transpose` *Transpose pitch*

Description

Transpose pitch by a number of semitones.

Usage

```
transpose(notes, n = 0, octaves = NULL, accidentals = NULL, key = NULL)

tp(notes, n = 0, octaves = NULL, accidentals = NULL, key = NULL)
```

Arguments

`notes`	character, a noteworthy string.
`n`	integer, positive or negative number of semitones to transpose.
`octaves`	`NULL` or character, `"tick"` or `"integer"` octave numbering in result.
`accidentals`	`NULL` or character, represent accidentals, `"flat"` or `"sharp"`.
`key`	`NULL` or character, use a key signature to specify and override `accidentals`. Ignored if `c` or `am`.

Details

This function transposes the pitch of notes in a noteworthy string.

Transposing is not currently supported on a phrase object. The notes in a phrase object have already been transformed to LilyPond syntax and mixed with other potentially complex information. Transposing is intended to be done on a string of notes prior to passing it to `phrase()`. It will work on strings that use either integer or tick mark octave numbering formats and flats or sharps, in any combination. The transposed result conforms according to the function arguments. When integer octaves are returned, all `3`s are dropped since the third octave is implicit in LilyPond.

When `octaves`, `accidentals` and `key` are `NULL`, formatting is inferred from `notes`. When mixed formats are present, tick format is the default for octave numbering and flats are the default for accidentals.

Value

character

Examples

```
transpose("a_3 b_4 c5", 0)
tp("a_3 b_4 c5", -1)
tp("a_3 b_4 c5", 1)
tp("a#3 b4 c#5", 11)
tp("a#3 b4 c#5", 12)
tp("r s a#3 b4 c#5", 13)
tp("a b' c#''", 2, "integer", "flat")
tp("a, b ceg", 2, "tick", "sharp")
```

---

tunings *Predefined instrument tunings*

Description

A data frame containing some predefined instrument tunings commonly used for guitar, bass, mandolin, banjo, ukulele and orchestral instruments.

Usage

```
tunings
```

Format

A data frame with 2 columns for the tuning ID and corresponding pitches and 32 rows for all predefined tunings.

---

tuplet *Tuplets*

Description

Helper function for generating tuplet syntax.

Usage

```
tuplet(x, n, string = NULL, a = 3, b = 2)

triplet(x, n, string = NULL)
```

Arguments

x	noteworthy string or phrase object.
n	integer, duration of each tuplet note, e.g., 8 for 8th note tuplet.
string	character, optional string or vector with same number of timesteps as x that specifies which strings to play for each specific note. Only applies when x is a noteworthy string.
a	integer, notes per tuplet.
b	integer, beats per tuplet.

Details

This function gives control over tuplet construction. The default arguments a = 3 and b = 2 generates a triplet where three triplet notes, each lasting for two thirds of a beat, take up two beats. n} is used to describe the beat duration with the same fraction-of-measure denominator notation used for phrases, e.g., 16th note triplet, 8th note triplet, etc.

If you provide a note sequence for multiple tuplets in a row of the same type, they will be connected automatically. It is not necessary to call tuplet() each time when the pattern is constant. If you provide a complete phrase object, it will simply be wrapped in the tuplet tag, so take care to ensure the phrase contents make sense as part of a tuplet.

Value

phrase

Examples

```
tuplet("c c# d", 8)
triplet("c c# d", 8)
tuplet("c c# d c c# d", 4, a = 6, b = 4)

p1 <- phrase("c c# d", "8-. 8( 8)", "5*3")
tuplet(p1, 8)
```

---

valid-noteinfo *Check note info validity*

Description

Check whether a note info string is comprised exclusively of valid note info syntax. noteinfo returns a scalar logical result indicating whether the entire set contains exclusively valid entries.

Usage

```
informable(x, na.rm = FALSE)

as_noteinfo(x, format = NULL)

is_noteinfo(x)
```

Arguments

x	character, a note info string.
na.rm	remove NAs.
format	NULL or character, the timestep delimiter format, "space" or "vector".

Details

as_noteinfo() can be used to coerce to the noteinfo class. Coercion will fail if the string is has any syntax that is not valid for note info. Using the noteinfo class is generally not needed by the user during an interactive session, but is available and offers its own print() and summary() methods for note info strings. The class is often used by other functions, and functions that output a note info string attach the noteinfo class.

When format = NULL, the timestep delimiter format is inferred from the note info string input. When unclear, such as with phrase objects, the default is space-delimited time.

Value

depends on the function

See Also

noteinfo(), valid-notes()

Examples

```
a <- notate("8x", "Start here")
x <- paste(a, "8[stacatto] 8-. 16 4.. 16- 16 2^ 2 4. 8( 4)( 4) 8*4 1 1")

informable(x) # is it of 'noteinfo' class; a validity check for any string
x <- as_noteinfo(x) # coerce to 'noteinfo' class
is_noteinfo(x) # check for 'noteinfo' class
x

summary(x)
```

---

valid-notes *Check note and chord validity*

Description

Check if a string is comprised exclusively of valid note and/or chord syntax.

Usage

```
is_note(x, na.rm = FALSE)

is_chord(x, na.rm = FALSE)

noteworthy(x, na.rm = FALSE)

as_noteworthy(x, octaves = NULL, accidentals = NULL, format = NULL)

is_noteworthy(x)
```

Arguments

x	character, a noteworthy string.
na.rm	remove NAs.
octaves	NULL or character, "tick" or "integer" octave numbering in result.
accidentals	NULL or character, represent accidentals, "flat" or "sharp".
format	NULL or character, the timestep delimiter format, "space" or "vector".

Details

is_note() and is_chord() are vectorized and their positive results are mutually exclusive. noteworthy() is also vectorized and performs both checks, but it returns a scalar logical result indicating whether the entire set contains exclusively valid entries.

as_noteworthy() can be used to coerce to the noteworthy class. Coercion will fail if the string is not noteworthy. While many functions will work on simple character strings and, if their syntax is valid, coerce them to the 'noteworthy' class, it is recommended to use this class. Not all functions are so aggressive, and several generic methods are implemented for the class. It also offers its own print() and summary() methods for noteworthy strings. An added benefit to using as_noteworthy() is to conform all notes in a noteworthy string to specific formatting for accidentals and octave numbering. Functions that output a noteworthy string attach the noteworthy class.

When octaves, accidentals, and format are NULL, formatting is inferred from the noteworthy string input. When mixed formats are present, tick format is the default for octave numbering and flats are the default for accidentals.

Value

depends on the function

See Also

note-checks(), note-metadata(), note-summaries(), note-coerce()

Examples

```
x <- "a# b_ c, d'' e3 g_4 A m c2e_2g2 cegh" # includes invalid syntax
data.frame(
  x = strsplit(x, " ")[[1]],
  note = is_note(x),
  chord = is_chord(x),
  either = noteworthy(x))

is_diatonic("ace ac#e d e_", "c")

x <- "a# b_ c,~ c, d'' e3 g_4 c2e_2g2"
noteworthy(x) # is it noteworthy; a validity check for any string
x <- as_noteworthy(x) # coerce to 'noteworthy' class, conform formatting
is_noteworthy(x) # check for 'noteworthy' class
x

summary(x)

x <- as_noteworthy(x, format = "vector", octaves = "integer",
                   accidentals = "flat")
x

summary(x)
```

Index