

# Package ‘tabr’

October 7, 2019

**Version** 0.3.5

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

**Description** 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.

The package also provides API wrapper functions for transcribing musical representations in R into guitar tablature (“`tabs”) using the ‘LilyPond’ backend (<<http://lilypond.org>>).

‘LilyPond’ is open source music engraving software for generating high quality sheet music based on markup syntax.

‘tabr’ generates ‘LilyPond’ files from R code and can pass them to ‘LilyPond’ to be rendered into sheet music pdf files from R.

While ‘LilyPond’ caters to sheet music in general and ‘tabr’ can be used to create basic sheet music, the transcription functions focus on leveraging ‘LilyPond’ specifically for creating quality guitar tablature.

The package offers nominal MIDI file output support in conjunction with rendering tablature output. See the ‘tuneR’ package for more general use of MIDI files in R.

**License** MIT + file LICENSE

**Encoding** UTF-8

**LazyData** true

**ByteCompile** true

**URL** <https://github.com/leonawicz/tabr>

**BugReports** <https://github.com/leonawicz/tabr/issues>

**Depends** R (>= 2.10)

**SystemRequirements** LilyPond

**Suggests** testthat, knitr, rmarkdown, covr, kableExtra, htmltools,  
fansi

**Imports** magrittr, dplyr, purrr, tidyr, crayon, ggplot2

**VignetteBuilder** knitr

**RoxygenNote** 6.1.1

**NeedsCompilation** no

**Author** Matthew Leonawicz [aut, cre]

**Maintainer** Matthew Leonawicz <matt\_leonawicz@esource.com>

**Repository** CRAN

**Date/Publication** 2019-10-06 22:00:02 UTC

## R topics documented:

append_phrases . . . . .	3
chord-compare . . . . .	4
chord-filter . . . . .	5
chord-mapping . . . . .	6
chords . . . . .	7
chord_arpeggiate . . . . .	11
chord_break . . . . .	12
chord_def . . . . .	12
chord_invert . . . . .	13
chord_is_diatonic . . . . .	14
chord_is_major . . . . .	15
chord_set . . . . .	16
dyad . . . . .	17
fretboard_plot . . . . .	18
guitarChords . . . . .	19
hp . . . . .	20
interval-helpers . . . . .	20
interval_semitones . . . . .	21
keys . . . . .	22
lilypond . . . . .	23
lp_chord_id . . . . .	25
mainIntervals . . . . .	26
midily . . . . .	26
miditab . . . . .	28
mode-helpers . . . . .	29
notate . . . . .	31
note-equivalence . . . . .	31
note-metadata . . . . .	33
note_is_natural . . . . .	35
note_slice . . . . .	37
phrase . . . . .	38
phrase-checks . . . . .	40
pitch_freq . . . . .	42
repeats . . . . .	43
rest . . . . .	45
scale-deg . . . . .	45
scale-helpers . . . . .	47
scale_chords . . . . .	48
score . . . . .	49
sf_phrase . . . . .	50

tab . . . . .	52
tabr . . . . .	54
tabrSyntax . . . . .	54
tabr_dep . . . . .	55
tabr_options . . . . .	55
tie . . . . .	56
track . . . . .	56
trackbind . . . . .	58
transpose . . . . .	59
tunings . . . . .	60
tuplet . . . . .	60
valid-notes . . . . .	61

**Index** **64**

append\_phrases                      *Concatenate and repeat*

**Description**

Helper functions for concatenating musical phrases and other strings together as well as repetition. The functions pc and pn are based on base functions paste and rep, respectively, but are tailored for efficiency in creating musical phrases.

**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**

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"))
```

---

chord-compare

*Rank, order and sort chords and notes*

---

**Description**

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

**Usage**

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

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

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

**Arguments**

chords	character, a noteworthy string, may include individual notes and chords.
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(chords)

chord_top(chords)

chord_slice(chords, index)
```

**Arguments**

chords	character, a noteworthy string, may include individual notes and chords.
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 if `index` is completely out of bounds for all chords.

**Value**

a noteworthy string

**Examples**

```
x <- "a2 ceg 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_fret = NA, min_fret = NA, bass_string = NA,
        open = NA, key = "c", ignore_octave = FALSE)

gc_fretboard(name, root_fret = NA, min_fret = NA, bass_string = NA,
             open = NA, key = "c", ignore_octave = FALSE)

gc_notes(name, root_fret = NA, min_fret = NA, bass_string = NA,
         open = NA, key = "c", ignore_octave = FALSE)

chord_is_known(notes)

chord_name_split(name)

chord_name_root(name)

chord_name_mod(name)
```

**Arguments**

name	character, chord name in tabr format, e.g., "bM b_m b_m7#5", etc.
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. NA 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 `chord_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.

`chord_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. `chord_name_root` and `chord_name_mod` are wrappers around this.

## Value

various, see details regarding each function.

## Examples

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

chord_name_root("a aM b_,m7#5")
chord_name_mod("a aM b_,m7#5")

gc_info("a") # a major chord, not a single note
gc_info("ceg a#m7_5") # only third 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,", 0:3)

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

## Description

These functions construct basic chord string notation from root notes.

**Usage**

```
chord_min(notes, key = "c", style = "default")
chord_maj(notes, key = "c", style = "default")
chord_min7(notes, key = "c", style = "default")
chord_dom7(notes, key = "c", style = "default")
chord_7s5(notes, key = "c", style = "default")
chord_maj7(notes, key = "c", style = "default")
chord_min6(notes, key = "c", style = "default")
chord_maj6(notes, key = "c", style = "default")
chord_dim(notes, key = "c", style = "default")
chord_dim7(notes, key = "c", style = "default")
chord_m7b5(notes, key = "c", style = "default")
chord_aug(notes, key = "c", style = "default")
chord_5(notes, key = "c", style = "default")
chord_sus2(notes, key = "c", style = "default")
chord_sus4(notes, key = "c", style = "default")
chord_dom9(notes, key = "c", style = "default")
chord_7s9(notes, key = "c", style = "default")
chord_maj9(notes, key = "c", style = "default")
chord_add9(notes, key = "c", style = "default")
chord_min9(notes, key = "c", style = "default")
chord_madd9(notes, key = "c", style = "default")
chord_min11(notes, key = "c", style = "default")
chord_7s11(notes, key = "c", style = "default")
chord_maj7s11(notes, key = "c", style = "default")
```



```
chord_11(notes, key = "c", style = "default")
chord_maj11(notes, key = "c", style = "default")
chord_13(notes, key = "c", style = "default")
chord_min13(notes, key = "c", style = "default")
chord_maj13(notes, key = "c", style = "default")
xm(notes, key = "c", style = "default")
xM(notes, key = "c", style = "default")
xm7(notes, key = "c", style = "default")
x7(notes, key = "c", style = "default")
x7s5(notes, key = "c", style = "default")
xM7(notes, key = "c", style = "default")
xm6(notes, key = "c", style = "default")
xM6(notes, key = "c", style = "default")
xdim(notes, key = "c", style = "default")
xdim7(notes, key = "c", style = "default")
xm7b5(notes, key = "c", style = "default")
xaug(notes, key = "c", style = "default")
x5(notes, key = "c", style = "default")
xs2(notes, key = "c", style = "default")
xs4(notes, key = "c", style = "default")
x9(notes, key = "c", style = "default")
x7s9(notes, key = "c", style = "default")
xM9(notes, key = "c", style = "default")
xadd9(notes, key = "c", style = "default")
```

```
xm9(notes, key = "c", style = "default")
xma9(notes, key = "c", style = "default")
xm11(notes, key = "c", style = "default")
x7s11(notes, key = "c", style = "default")
xM7s11(notes, key = "c", style = "default")
x_11(notes, key = "c", style = "default")
xM11(notes, key = "c", style = "default")
x_13(notes, key = "c", style = "default")
xm13(notes, key = "c", style = "default")
xM13(notes, key = "c", style = "default")
```

### Arguments

notes	character, a noteworthy string of chord root notes.
key	key signature. See details.
style	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	<i>Arpeggiate a chord</i>
------------------	---------------------------

---

### 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. See examples.

**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 tabl notation. See `id` column in `guitar Chords` 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**

<code>chord</code>	character, a single chord.
<code>n</code>	inversion.
<code>limit</code>	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. Its 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)
```

---

<code>chord_is_diatonic</code>	<i>Check if chords are diatonic</i>
--------------------------------	-------------------------------------

---

**Description**

Check whether chords are diatonic in a given key.

**Usage**

```
chord_is_diatonic(chord, key = "c")
```

**Arguments**

<code>chord</code>	character, a chord string. May be a vector.
<code>key</code>	character, key signature.

**Details**

This function strictly accepts chord strings. To check if notes are in a scale, see [note\\_in\\_scale](#). To check generally if a noteworthy string is fully diatonic, see [is\\_diatonic](#).

**Value**

logical

**See Also**

[note\\_in\\_scale](#), [is\\_diatonic](#)

**Examples**

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

---

chord\_is\_major

*Check if chords are major or minor*

---

**Description**

Check if chords are major or minor where possible.

**Usage**

```
chord_is_major(chords)
```

```
chord_is_minor(chords)
```

**Arguments**

chords            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	<i>Generate a chord set</i>
-----------	-----------------------------

---

### Description

Generate a chord set for a music score.

### Usage

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

### Arguments

x	character, n-string chord description from lowest to highest pitch, strings n through 1. E.g., "xo221o". See details.
id	character, the name of the chord in LilyPond readable format, e.g., "a:m". Ignored if x is already a named vector.

### 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("xx997x", "x5553x", "x7775x", "ooo22o", "232oxx", NA, NA)
chord_set(chord_position, chord_names)
```



---

dyad	<i>Construct a dyad</i>
------	-------------------------

---

**Description**

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

**Usage**

```
dyad(notes, interval, reverse = FALSE, key = "c")
```

**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.
key	character, key signature.

**Details**

The interval may be specified by semitones or 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")
sapply(x, function(x) dyad("a", x))
sapply(x, function(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")
```

---

fretboard_plot	<i>Fretboard plot</i>
----------------	-----------------------

---

### Description

Create a fretboard diagram.

### Usage

```
fretboard_plot(string, fret, labels = NULL, mute = FALSE,
  label_size = 4, label_color = "white", point_size = 10,
  point_color = "black", point_fill = "black", group = NULL,
  horizontal = FALSE, left_handed = FALSE, fret_range = NULL,
  key = "c", tuning = "standard", show_tuning = FALSE)
```

### Arguments

string	integer or as tabl-style character string, string numbers.
fret	integer or as tabl-style character string, fret numbers.
labels	character, optional text labels, must be one for every point.
mute	logical, whether to mute notes, typically a vector corresponding to string and fret.
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.
key	character, key signature, used to enforce type of accidentals when labels = "notes".
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.

### Details

This function is under development and subject to change.

Create a fretboard diagram ggplot object. 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".

**Value**

a ggplot object

**Examples**

```
# open chord
am_frets <- c(0, 0, 2, 2, 1, 0) # first note will be muted; 'x' is drawn at 0
idx <- c(1, 1, 2, 2, 2, 1)
fill <- c("white", "black")[idx]
lab_col <- c("black", "white")[idx]
mute <- c(TRUE, rep(FALSE, 5))
fretboard_plot(6:1, am_frets, "notes", mute, label_color = lab_col,
               point_fill = fill)

# moveable chord
fretboard_plot(6:1, am_frets, mute = mute, point_fill = fill,
               fret_range = c(0, 4), horizontal = TRUE, show_tuning = TRUE)

# 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
fretboard_plot(string, fret, labels = "notes")
```

---

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")
```

---

interval-helpers *Interval helpers*

---

### Description

Helper functions for musical intervals defined by two notes.

### Usage

```
pitch_interval(note1, note2, ignore_octave = FALSE)

scale_interval(note1, note2, format = c("mmp_abb", "mmp", "ad_abb",
    "ad"), ignore_octave = FALSE)

tuning_intervals(tuning = "standard")
```

**Arguments**

note1	character, first note. Must be a single note.
note2	character, second note.
ignore_octave	logical, reduce the interval to that defined by notes within a single octave.
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**

Intervals are directional. `pitch_interval` returns the number of semitones defining the distance between two notes. The interval is negative if `note1` has higher pitch than `note2`. For `scale_interval`, a character string is returned that provides the named main interval, simple or compound, defined by the two notes. This function will return NA for any uncommon interval not listed in [mainIntervals](#).

**Value**

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

**See Also**

[mainIntervals](#)

**Examples**

```
pitch_interval("b", "c4")
pitch_interval("c", "d")
scale_interval("c", "e_")
```

---

interval_semitones	<i>Interval semitones</i>
--------------------	---------------------------

---

**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)
```

---

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, string_names = NULL, paper = NULL, endbar = TRUE,
  midi = TRUE, path = NULL)
```

**Arguments**

<code>score</code>	a score object.
<code>file</code>	character, LilyPond output file ending in <code>.ly</code> . May include an absolute or relative path.
<code>key</code>	character, key signature, e.g., <code>c</code> , <code>b_</code> , <code>f#m</code> , etc.
<code>time</code>	character, defaults to <code>"4/4"</code> .
<code>tempo</code>	character, defaults to <code>"2 = 60"</code> .
<code>header</code>	a named list of arguments passed to the header of the LilyPond file. See details.
<code>string_names</code>	label strings at beginning of tab staff. <code>NULL</code> (default) for non-standard tunings only, <code>TRUE</code> or <code>FALSE</code> for force on or off completely.

paper	a named list of arguments for the LilyPond file page layout. See details.
endbar	character, the end bar.
midi	logical, add midi inclusion specification to LilyPond file.
path	character, optional output directory prefixed to file, may be an absolute or relative path. If NULL (default), only file is used.

### Details

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

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

All paper list elements are numeric except `page_numbers`, which is logical. The options for paper include:

- `textheight`
- `linewidth`
- `indent`
- `first_page_number`
- `page_numbers`
- `fontsize`

### Value

nothing returned; a file is written.

### See Also

[tab](#), [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)
```



---

lp_chord_id	<i>LilyPond chord notation</i>
-------------	--------------------------------

---

**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, tabl format chord name.
exact	logical, return a more exact LilyPond chord representation.
...	additional arguments passed to transpose.

**Details**

These functions take a tabl 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 tabl 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)
```

---

mainIntervals	<i>Main musical intervals.</i>
---------------	--------------------------------

---

**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	<i>Convert MIDI to LilyPond file</i>
--------	--------------------------------------

---

**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, path = NULL)
```

**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.
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, verbose detail.
lyric	logical, treat all text as lyrics.
path	character, optional output directory prefixed to file, may be an absolute or relative path. If NULL (default), only file is used.

## 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! The next version of this function will add a default argument `midi_out = FALSE` to remove this from the generated LilyPond file. If `TRUE` and the basename of `midi_file` matches the basename of `file`, then `file` will be renamed, the basename appended with a `-1`.

`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 successfully set as follows. On package load, `tabr` will attempt to check for `midi2ly.exe` at `C:/Program Files (x86)/LilyPond/usr/bin/midi2ly.py` and similarly for the `python.exe` that ships with LilyPond at `C:/Program Files (x86)/LilyPond/usr/bin/python.exe`. If this is not where LilyPond is installed, then LilyPond and Python need to be provided to `tabr_options` or 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	<i>Convert MIDI to tablature</i>
---------	----------------------------------

---

### Description

Convert a MIDI file to sheet music/guitar tablature.

### Usage

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

### 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.
path	character, optional output directory prefixed to file, may be an absolute or relative path. If NULL (default), only file is used.
details	logical, set to FALSE to disable printing of log output to console.
...	additional arguments passed to <a href="#">midily</a> .

### 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! The next version of this function will add a default argument `midi_out = FALSE` to remove this from the generated LilyPond file. If TRUE and the basename of `midi_file` matches the basename of `file`, then `file` will be renamed, the basename appended with a -1.

On Windows systems, it may be necessary to specify a path in `tabr_options` to both `midi2ly` and `python` if they are not already successfully set as follows. On package load, `tabr` will attempt to check for `midi2ly.exe` at `C:/Program Files (x86)/LilyPond/usr/bin/midi2ly.py` and similarly for the `python.exe` that ships with LilyPond at `C:/Program Files (x86)/LilyPond/usr/bin/python.exe`. If this is not where LilyPond is installed, then LilyPond and Python need to be provided to `tabr_options` or 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))

```

---

notate	<i>Add text to music staff</i>
--------	--------------------------------

---

**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-equivalence	<i>Note, pitch and chord equivalence</i>
------------------	--

---

**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-metadata

*Noteworthy string metadata*

---

### Description

Inspect metadata for noteworthy strings.

### Usage

`n_steps(notes)`

`n_notes(notes)`

n\_chords(notes)  
n\_octaves(notes)  
tally\_notes(notes)  
tally\_pitches(notes)  
tally\_octaves(notes)  
distinct\_notes(notes)  
distinct\_pitches(notes)  
distinct\_octaves(notes)  
pitch\_range(notes)  
semitone\_range(notes)  
semitone\_span(notes)  
octave\_range(notes)  
octave\_span(notes)  
octave\_type(notes)  
accidental\_type(notes)  
time\_format(notes)  
is\_space\_time(notes)  
is\_vector\_time(notes)

### Arguments

notes            character, a noteworthy string, space-delimited or vector of individual entries.

### Details

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. `distinct_notes` and `distinct_pitches` filter a noteworthy string to its unique elements, respectively. These functions return another noteworthy string.

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

`*_span` functions are just the size of a range, e.g., `semitone_range` and `semitone_span`.

Functions pertaining to type or format of a noteworthy string provide information on how a particular string is defined, e.g. `time_format`.

## Value

integer or character

## Examples

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

tally_notes(x)
tally_pitches(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)

octave_type(x)
accidental_type(x)
time_format(x)
is_space_time(x)
is_vector_time(x)
```

---

note\_is\_natural

*Inspect or coerce noteworthy string formats*

---

## Description

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

**Usage**

```

note_is_natural(notes)

note_is_accidental(notes)

note_is_flat(notes)

note_is_sharp(notes)

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

sharpen_flat(notes, ignore_octave = FALSE)

flatten_sharp(notes, ignore_octave = FALSE)

note_set_key(notes, key = "c")

as_tick_octaves(notes)

as_integer_octaves(notes)

as_space_time(notes)

as_vector_time(notes)

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.
ignore_octave	logical, strip any octave notation that may be present, returning only the basic notes without explicit pitch.
key	character, key signature to coerce any accidentals to the appropriate form for the key. May also specify "sharp" or "flat".

**Details**

The `note_is_*` functions strictly allow individual notes, not chords. The other functions listed here accept any noteworthy string including those containing chords.

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

## Examples

```
x <- "a_2 a a#"
note_is_natural(x)
note_is_accidental(x)
note_is_flat(x)
note_is_sharp(x)

x <- "e_2 a_, b_, c##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\_slice

*Slice, rotate, shift and arpeggiate notes*

---

## Description

Helper functions for indexing and moving notes within noteworthy strings.

## Usage

```
note_slice(notes, ...)

note_rotate(notes, n = 0)

note_shift(notes, n = 0)

note_arpeggiate(notes, n = 0, ...)
```

**Arguments**

notes	character, a noteworthy string, space-delimited or vector of individual entries.
...	For note_slice, an integer or logical vector. See details. For note_arpeggiate, additional arguments to transpose, specifically key and style.
n	integer, degree of rotation.

**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\_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, as if inverting a broken chord. In this case notes is strictly interpreted and may not include chords. Octave numbering applies, though large multi-octave gaps will be condensed in the process. Given the context of note\_shift, the notes sequence should be ordered by increasing pitch. If it is not, ordering will be forced with each inversion during the n shifts.

note\_arpeggiate also allows notes only. It is similar to note\_shift, except that instead of a moving window, it grows from the original set of notes by n in the direction of the sign of n.

**Value**

character

**Examples**

```
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", 5)
note_arpeggiate("c e g", -5)
```

---

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, string = NULL, bar = FALSE)
```

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

**Arguments**

notes	character, notes a through g, comprising a noteworthy string. notes. See details.
info	character, metadata pertaining to the notes . See details.
string	character, optional string that specifies which guitar strings to play for each specific note.
bar	logical, insert a bar check at the end of the phrase.

**Details**

Meeting all of the requirements for a string of notes to be valid `tabr` syntax is referred to as *noteworthy*. Noteworthy strings are referred to throughout the documentation. Such requirements are outlined below.

Noteworthy strings use space-delimited time. This means that notes and chords separated in time are separated in the notes string by spaces. This is by far the most common usage. However, many functions in `tabr`, including `phrase`, allow a notes or similar first function argument to be provided in vector form where each vector element is a single note or chord (single point in time). Internally, functions like `phrase` will manipulate these forms back and forth as needed. Having both input options provides useful flexibility for music programming in `tabr` in general. The pipe operator can also be leveraged to chain several functions together.

Sharps and flats are indicated by appending `#` and `_`, respectively, e.g. `f#` or `g_`.

Specifying notes that are one or multiple octaves below or above the third octave can be done by appending one or multiple commas or single quote tick marks, respectively, e.g. `c,` or `c''`. But this is not necessary. Instead, you can use octave numbering. This may be easier to read, generally more familiar, potentially requires less typing, can still be omitted completely for the third octave (no need to type `c3`, `d3`, ...), and is automatically converted for you by `phrase` to the tick mark format interpreted by `LilyPond`. That said, using the raised and lowered tick mark approach can be surprisingly easier to read for chords, which have no spaces between notes, especially six-string chords, given that the tick marks help break up the notes in the chord visually much more so than integers do. See examples.

The function `p` is a convenient shorthand wrapper for `phrase`.

Tied notes indicated by `~` are part of the note notation and not part of the `info` notation, e.g. `c''~`.

Notes can comprise chords. These are bound tightly rather than space-delimited, as they are not separated in time. For example, a C chord could be given as `ceg` and in the case of tied notes would be `c~e~g~`.

Other information about a note is indicated with the `info` string. The most pertinent information, minimally required, is the note duration. A string of space-delimited notes will always be accompanied by a space-delimited string of an equal number of integer durations. Durations are powers of 2: 1, 2, 4, 8, 16, 32, 64. They represent the fraction of a measure, e.g., 2 means 1/2 of a measure

and 8 refers to an eighth note. Dotted notes are indicated by adding . immediately after the integer, e.g., 2. or 2. . . Any other note metadata is appended to these durations. See examples.

Opening and closing slurs (or hammer ons and pull offs) are indicated with opening and closing parentheses, slides with -, and simple bends with ^. Text annotations aligned vertically with a note in time on the staff is done by appending the text to the note info entry itself. See [notate](#). For more details and example, see the package vignettes.

## Value

a phrase.

## Examples

```
phrase("c ec'g' ec'g'", "4 4 2") # no string numbers (not recommended)
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 1 1", "5 432 432") # same as above
```

---

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)

phrase_strings(phrase, collapse = TRUE)

notable(phrase)
```

## Arguments

phrase	phrase object or character string (candidate phrase).
collapse	logical, collapse result into a single string ready for phrase construction.



## 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 handle arbitrary text notations attached to notes.

`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-frequency conversion*


---

### Description

Convert between pitch and frequency.

### Usage

```

pitch_freq(notes, a4 = 440)

pitch_semitones(notes)

chord_freq(notes, a4 = 440)

chord_semitones(notes)

freq_pitch(freq, a4 = 440, collapse = FALSE, ...)

freq_semitones(freq, 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.
collapse	logical, collapse result into a single string.
...	other arguments passed to <code>transpose</code> , specifically <code>key</code> and <code>style</code> .

### 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**

numeric vector

**Examples**

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

freq_pitch(y)

identical(as_noteworthy(x), freq_pitch(y, collapse = TRUE))
```

---

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 or basic 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 <i>measure</i> repeats of more than two repeats ( $n > 2$ ).
step	integer, print the <i>measure</i> 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	list of phrases or basic strings, 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	<i>Create rests</i>
------	---------------------

---

**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	<i>Scale degrees and mappings</i>
-----------	-----------------------------------

---

**Description**

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

**Usage**

```
scale_degree(notes, key = "c", scale = "diatonic",  
             naturalize = FALSE, roman = FALSE, ...)  
  
scale_note(deg, key = "c", scale = "diatonic", collapse = FALSE, ...)  
  
note_in_scale(notes, key = "c", scale = "diatonic", ...)
```

**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 <code>scale_*</code> function.
naturalize	logical, whether to naturalize any sharps or flats before obtaining the scale degree.
roman	logical, return integer scale degrees as Roman numerals.
...	additional arguments passed to the scale function, e.g., <code>sharp = FALSE</code> for <code>scale_chromatic</code> .
deg	integer, roman class, or character roman, the scale degree.
collapse	logical, collapse result into a single string ready for phrase construction.

**Details**

Obtain the scale degree of a note in a supported scale with `scale_degree`. This function also works with chords inside noteworthy strings. It only considers the first note in each space-delimited entry. notes may be a vector of single entries (non-delimited). Notes return NA if not explicitly in the scale. This includes cases where the note is in the scale but is notated as sharp or flat when the scale and/or key provide the opposite.

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.

`note_in_scale` performs a vectorized logical check if each note is in a given scale. This function strictly accepts single notes in noteworthy strings. To check if chords are diatonic to the scale, see [chord\\_is\\_diatonic](#). To check generally if a noteworthy string is fully diatonic, see [is\\_diatonic](#).

**Value**

integer, or roman class if `roman = TRUE` for `scale_degree`. character for `scale_note`.

**See Also**

[scale-helpers](#)

**Examples**

```
scale_degree("c e g")
scale_degree("c e g", roman = TRUE)
scale_degree("c e g", key = "d")
scale_degree("c, e_3 g' f#ac#")
scale_degree("c, e_3 g' f#ac#", naturalize = TRUE)
scale_degree("c, e_3 g' f#ac#", scale = "chromatic")
scale_degree("c, e_3 g' f#ac#", scale = "chromatic", sharp = FALSE)

scale_note(1:3)
```

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

---

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	<i>Create a music score</i>
-------	-----------------------------

---

**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 <code>chord_set</code> , 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 score table.

**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](#). It 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.

### Usage

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

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

```
sf_note(...)
```

```
sfn(...)
```

### Arguments

string	character, string numbers associated with notes, or provide all information here and ignore fret and info. See details.
fret	character, fret numbers associated with notes.
info	character, metadata associated with notes.
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	logical, insert a bar check at the end of the phrase.
...	arguments passed to <code>sf_phrase</code> .

### Details

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 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("654321 6s 12 1 21", "133211 355333 11 (13) (13)(13)", "4 4 8 8 4",
         key = "f")
sfp("6s*2 1*4", "000232*2 2*4", "4 4 8*4", tuning = "dropD", key = "d")

# compare with single-argument input
s <- "3s*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)
```

---

tab

*Create tablature*

---

## Description

Create sheet music/guitar tablature from a music score.

## Usage

```
tab(score, file, key = "c", time = "4/4", tempo = "2 = 60",
     header = NULL, string_names = NULL, paper = NULL, endbar = TRUE,
     midi = TRUE, keep_ly = FALSE, path = NULL, details = TRUE)
```

**Arguments**

score	a score object.
file	character, output file ending in .pdf or .png. 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".
header	a named list of arguments passed to the header of the LilyPond file. 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.
paper	a named list of arguments for the LilyPond file page layout. See details.
endbar	character, the end bar.
midi	logical, output midi file in addition to tablature.
keep_ly	logical, keep LilyPond file.
path	character, optional output directory prefixed to file, may be an absolute or relative path. If NULL (default), only file is used.
details	logical, set to FALSE to disable printing of log output to console.

**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.

For Windows users, add the path to the LilyPond executable to the system path variable. For example, if the file is at C:/Program Files (x86)/LilyPond/usr/bin/lilypond.exe, then add C:/Program Files (x86)/LilyPond/usr/bin to the system path.

**Value**

nothing returned; a file is written.

**See Also**

[lilypond](#), [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, details = FALSE) # requires LilyPond installation
}
```

---

tabr	<i>tabr: Music notation syntax, manipulation, analysis and transcription in R.</i>
------	--

---

### 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 package also provides API wrapper functions for transcribing musical representations in R into guitar tablature ("tabs") using the 'LilyPond' backend <<http://lilypond.org>>.

### Details

LilyPond is an open source music engraving program for generating high quality sheet music based on markup syntax. `tabr` generates LilyPond files from R code and can pass them to LilyPond to be rendered into sheet music pdf files from R.

While LilyPond caters to sheet music in general and `tabr` can be used to create basic sheet music, the transcription functions focus on leveraging LilyPond specifically for creating quality guitar tablature.

`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 many of its functions without installing LilyPond if you do not intend to render tabs.

---

<code>tabrSyntax</code>	<i>tabr syntax.</i>
-------------------------	---------------------

---

### 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_dep	<i>Deprecated</i>
----------	-------------------

---

**Description**

Deprecated tabr functions.

**Usage**

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

```
glue(...)
```

**Arguments**

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

---

tabr_options	<i>Options</i>
--------------	----------------

---

**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.py 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:/Program Files (x86)/LilyPond/usr/bin/lilypond.exe"
tabr_options(lilypond = lilypond_path)

```

---

tie	<i>Tied notes</i>
-----	-------------------

---

**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'")
```

---

track	<i>Create a music track</i>
-------	-----------------------------

---

**Description**

Create a music track from a collection of musical phrases.

**Usage**

```

track(phrase, tuning = "standard", voice = 1L,
      music_staff = "treble_8", ms_transpose = 0, ms_key = NA,
      no_tab = FALSE)

```



**Arguments**

phrase	a phrase object.
tuning	character, space-delimited pitches describing the instrument string tuning or a predefined tuning ID (see <a href="#">tunings</a> ). Defaults to standard guitar tuning. Tick or integer octave numbering accepted for custom tuning entries.
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).
music_staff	add a standard sheet music staff above the tablature staff. See details.
ms_transpose	integer, positive or negative number of semitones to transpose an included music staff relative to the tablature staff. See details.
ms_key	character, specify the new key signature for a transposed music staff. See details.
no_tab	logical, suppress the default guitar tablature so that only a standard music staff is associated with the track. Ignored if <code>music_staff = NA</code> .

**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 simply row binding that properly preserves phrase and track classes.

The default for an additional music staff is "treble\_8" for 8va treble clef, which is commonly displayed in quality guitar tablature above the tablature staff to include precise rhythm and timing information. Note that guitar is a transposing instrument. For this reason, the default ID is "treble\_8", not "treble". Set `music_staff = NA` to suppress the additional music staff above the tablature staff. This is appropriate for simple patterns where there are already multiple tracks and the additional space required for two staves per instrument is unnecessary and wasteful.

The arguments `ms_transpose` and `ms_key` pertain to the transposition of the music staff relative to the tab staff if `music_staff` is not NA. These arguments default to 0 and NA, respectively. The transposition and new key are simply stored in the `ms_transpose` and `ms_key` columns in the resulting track table. This information is used by `lilypond` or `tab` to transpose the music staff relative to the tab staff at the time of LilyPond file generation. Non-zero semitone transposition works without providing an explicit new key signature, but it is recommended to specify because it helps ensure the correct selection of accidentals in the output. As with the [transpose](#) function, you can simply specify `key = "flat"` or `key = "sharp"`. The exact key signature is not required; it is merely more clear and informative for the user.

**Value**

a track table.

**Examples**

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

---

 trackbind

*Bind track tables*


---

### Description

Bind together track tables by row.

### Usage

```
trackbind(..., tabstaff)
```

### Arguments

...                    track tables.  
 tabstaff              integer, ID vector indicating the tablature staff for each track. See details.

### Details

This function appends multiple track tables into a single track table for preparation of generating a multi-track score. `tabstaff` is used to separate music 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, the `tabstaff` ID automatically propagates `1:n` for `n` tracks passed to `...` when binding these tracks together. This expresses the default assumption of one tab staff per track. This is the typical use case where each single track object being bound into a multi-track object is a fully separated track on its own staff.

However, some tracks represent different voices that share the same staff. These should be assigned the same staff ID value, in which case you must provide the `tabstaff` argument. An error will be thrown if any two tracks have both the same voice and the same `tabstaff`. The pair must be unique. E.g., provide `tabstaff = 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 track table.

### 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, tabstaff = c(1, 1))
```

---

transpose	<i>Transpose pitch</i>
-----------	------------------------

---

### Description

Transpose pitch by a number of semitones.

### Usage

```
transpose(notes, n = 0, key = NA, style = c("default", "tick",
  "integer", "strip"))
```

```
tp(notes, n = 0, key = NA, style = c("default", "tick", "integer",
  "strip"))
```

### Arguments

notes	character, a noteworthy string.
n	integer, positive or negative number of semitones to transpose.
key	character, the new key signature after transposing notes. See details.
style	character, specify tick or integer style octave numbering in result. See details.

### Details

This function transposes the pitch of notes in a valid character string. The string must be of the form passed to the `info` argument to [phrase](#).

Transposing is not done on a phrase object. The notes in a phrase object have already been transformed to LilyPond syntax and mixed with other potentially complex and variable 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. The transposed result will be a string with integer octave numbering.

If `key` is provided, this helps ensure proper use of sharps vs. flats. Alternatively, you can simply provide `key = "sharp"` or `key = "flat"`. The exact key signature is not required, just more clear and informative for the user. If not provided (`key = NA`), transposition lacks full information and simply defaults to sharpening any resulting accidentals for positive `n` and flattening for negative `n`. `n = 0` returns the input without any modification.

The only element other pitch that occurs in a valid notes string is a rest, `"r"` or `"s"` (silent rest). Rests are ignored by `transpose`.

The default style is to use tick style if no integers occur in notes. The `"tick"` and `"integer"` options force the respective styles. When integer style is returned, all 3s are dropped since the third octave is the implicit center in LilyPond. `style = "strip"` removes any explicit octave information.

### 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("a#3 b4 c#5", 13)
tp("a3 b4 c5", 2, key = "f")
tp("a3 b4 c5", 2, key = "g")
tp("a b' c''", 2, key = "flat")
tp("a, b ceg", 2, key = "sharp")
```

---

tunings	<i>Predefined instrument tunings.</i>
---------	---------------------------------------

---

**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.

---

tuple	<i>Tuplets</i>
-------	----------------

---

**Description**

Helper function for generating tuple syntax.

**Usage**

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

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

**Arguments**

x	phrase object or character string of notes.
n	integer, duration of each tuplet note, e.g., 8 for 8th note tuplet.
string,	character, optional string that specifies which guitar strings to play for each specific note.
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 notes in `tabr` 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-notes

---

*Check note and chord validity*


---

**Description**

Check whether a string is comprised exclusively of valid note and/or chord substring syntax. `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.

**Usage**

```

is_note(x)

is_chord(x)

noteworthy(x)

is_diatonic(x, key = "c")

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

```

**Arguments**

x	character, space-delimited entries or a vector of single, non-delimited entries.
key	character, key signature.
format	NULL or character ("space" or "vector"), the timestep delimiter format.
octaves	NULL or character ("tick" or "integer"), the octave representation.
accidentals	NULL or character ("flat" or "sharp"), the accidentals representation.

**Details**

as\_noteworthy can be used to coerce to the noteworthy class. Coercion will fail if the string is not noteworthy. Using the noteworthy class is generally not needed by the user during an interactive session, but is available and offers its own print and summary methods for noteworthy strings. It is more likely to be used by other functions and functions that output a noteworthy string generally attach the noteworthy class.

is\_diatonic performs a vectorized logical check on a noteworthy string for all notes and chords. To check strictly notes or chords, see [note\\_in\\_scale](#) and [chord\\_is\\_diatonic](#).

**Value**

logical

**See Also**

[note\\_in\\_scale](#), [chord\\_is\\_diatonic](#)

**Examples**

```

x <- "a# b_ c, d'' e3 g_4 A m c2e_2g2 cegh"
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"  
x <- as_noteworthy(x)  
x
```

```
summary(x)
```

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

```
x
```

```
summary(x)
```

# Index

## \*Topic **datasets**

- guitarChords, 19
  - mainIntervals, 26
  - tabrSyntax, 54
  - tunings, 60
- accidental\_type (note-metadata), 33
- append\_phrases, 3
- as\_integer\_octaves (note\_is\_natural), 35
- as\_noteworthy, 37
- as\_noteworthy (valid-notes), 61
- as\_phrase (phrase-checks), 40
- as\_space\_time (note\_is\_natural), 35
- as\_tick\_octaves (note\_is\_natural), 35
- as\_vector\_time (note\_is\_natural), 35
- 
- chord-compare, 4
- chord-filter, 5
- chord-mapping, 6
- chord\_11 (chords), 7
- chord\_13 (chords), 7
- chord\_5 (chords), 7
- chord\_7s11 (chords), 7
- chord\_7s5 (chords), 7
- chord\_7s9 (chords), 7
- chord\_add9 (chords), 7
- chord\_arpeggiate, 11
- chord\_aug (chords), 7
- chord\_break, 12
- chord\_def, 12
- chord\_dim (chords), 7
- chord\_dim7 (chords), 7
- chord\_dom7 (chords), 7
- chord\_dom9 (chords), 7
- chord\_freq (pitch\_freq), 42
- chord\_invert, 13
- chord\_is\_diatonic, 14, 46, 62
- chord\_is\_known (chord-mapping), 6
- chord\_is\_major, 15
- chord\_is\_minor (chord\_is\_major), 15
- 
- chord\_m7b5 (chords), 7
- chord\_madd9 (chords), 7
- chord\_maj (chords), 7
- chord\_maj11 (chords), 7
- chord\_maj13 (chords), 7
- chord\_maj6 (chords), 7
- chord\_maj7 (chords), 7
- chord\_maj7s11 (chords), 7
- chord\_maj9 (chords), 7
- chord\_min (chords), 7
- chord\_min11 (chords), 7
- chord\_min13 (chords), 7
- chord\_min6 (chords), 7
- chord\_min7 (chords), 7
- chord\_min9 (chords), 7
- chord\_name\_mod (chord-mapping), 6
- chord\_name\_root (chord-mapping), 6
- chord\_name\_split (chord-mapping), 6
- chord\_order (chord-compare), 4
- chord\_rank (chord-compare), 4
- chord\_root (chord-filter), 5
- chord\_semitones, 13
- chord\_semitones (pitch\_freq), 42
- chord\_set, 16
- chord\_slice (chord-filter), 5
- chord\_sort (chord-compare), 4
- chord\_sus2 (chords), 7
- chord\_sus4 (chords), 7
- chord\_top (chord-filter), 5
- chords, 7
- 
- distinct\_notes (note-metadata), 33
- distinct\_octaves (note-metadata), 33
- distinct\_pitches (note-metadata), 33
- dup (tabr\_dep), 55
- dyad, 17
- 
- flatten\_sharp (note\_is\_natural), 35
- freq\_pitch (pitch\_freq), 42
- freq\_semitones (pitch\_freq), 42



- fretboard\_plot, 18
- gc\_fretboard (chord-mapping), 6
- gc\_info (chord-mapping), 6
- gc\_notes (chord-mapping), 6
- glue (tabr\_dep), 55
- guitarChords, 19
- hp, 20
- interval\_helpers, 20
- interval\_semitones, 21
- is\_chord (valid-notes), 61
- is\_diatonic, 14, 15, 46
- is\_diatonic (valid-notes), 61
- is\_mode (mode\_helpers), 29
- is\_note (valid-notes), 61
- is\_space\_time (note-metadata), 33
- is\_vector\_time (note-metadata), 33
- key\_is\_flat (keys), 22
- key\_is\_major (keys), 22
- key\_is\_minor (keys), 22
- key\_is\_natural (keys), 22
- key\_is\_sharp (keys), 22
- key\_n\_flats (keys), 22
- key\_n\_sharps (keys), 22
- keys, 22, 30, 47, 48
- lilypond, 23, 27, 29, 53
- lp\_chord\_id, 25
- lp\_chord\_mod (lp\_chord\_id), 25
- mainIntervals, 17, 21, 22, 26
- midily, 24, 26, 28, 29
- miditab, 27, 28, 53
- mode\_helpers, 29
- mode\_aeolian (mode\_helpers), 29
- mode\_dorian (mode\_helpers), 29
- mode\_ionian (mode\_helpers), 29
- mode\_locrian (mode\_helpers), 29
- mode\_lydian (mode\_helpers), 29
- mode\_mixolydian (mode\_helpers), 29
- mode\_modern (mode\_helpers), 29
- mode\_phrygian (mode\_helpers), 29
- mode\_rotate (mode\_helpers), 29
- modes (mode\_helpers), 29
- n\_chords (note-metadata), 33
- n\_notes (note-metadata), 33
- n\_octaves (note-metadata), 33
- n\_steps (note-metadata), 33
- naturalize (note\_is\_natural), 35
- notable (phrase-checks), 40
- notate, 31, 40
- note-equivalence, 31
- note-metadata, 33
- note\_arpeggiate (note\_slice), 37
- note\_in\_scale, 14, 15, 62
- note\_in\_scale (scale-deg), 45
- note\_is\_accidental (note\_is\_natural), 35
- note\_is\_equal (note-equivalence), 31
- note\_is\_flat (note\_is\_natural), 35
- note\_is\_identical (note-equivalence), 31
- note\_is\_natural, 35
- note\_is\_sharp (note\_is\_natural), 35
- note\_rotate (note\_slice), 37
- note\_set\_key (note\_is\_natural), 35
- note\_shift (note\_slice), 37
- note\_slice, 37
- noteworthy (valid-notes), 61
- notify (phrase-checks), 40
- octave\_is\_equal (note-equivalence), 31
- octave\_is\_identical (note-equivalence), 31
- octave\_range (note-metadata), 33
- octave\_span (note-metadata), 33
- octave\_type (note-metadata), 33
- p (phrase), 38
- pc (append\_phrases), 3
- pct (repeats), 43
- phrase, 38, 44, 50, 52, 57, 59
- phrase-checks, 40
- phrase\_info (phrase-checks), 40
- phrase\_notes (phrase-checks), 40
- phrase\_strings (phrase-checks), 40
- phrasey (phrase-checks), 40
- pitch\_freq, 42
- pitch\_interval (interval\_helpers), 20
- pitch\_is\_equal (note-equivalence), 31
- pitch\_is\_identical (note-equivalence), 31
- pitch\_range (note-metadata), 33
- pitch\_semitones (pitch\_freq), 42
- pn (append\_phrases), 3
- pretty\_notes (note\_is\_natural), 35

- repeats, 43
- rest, 45
- rp (repeats), 43
- scale-deg, 45
- scale-helpers, 47
- scale\_chords, 48
- scale\_chromatic (scale-helpers), 47
- scale\_degree (scale-deg), 45
- scale\_diatonic (scale-helpers), 47
- scale\_harmonic\_minor (scale-helpers), 47
- scale\_hungarian\_minor (scale-helpers), 47
- scale\_interval (interval-helpers), 20
- scale\_jazz\_minor (scale-helpers), 47
- scale\_major (scale-helpers), 47
- scale\_melodic\_minor (scale-helpers), 47
- scale\_minor (scale-helpers), 47
- scale\_note (scale-deg), 45
- score, 49
- semitone\_range (note-metadata), 33
- semitone\_span (note-metadata), 33
- sf\_note (sf\_phrase), 50
- sf\_phrase, 50
- sfn (sf\_phrase), 50
- sfp (sf\_phrase), 50
- sharpen\_flat (note\_is\_natural), 35
- tab, 24, 27, 29, 49, 52
- tabr, 54
- tabr-package (tabr), 54
- tabr\_dep, 55
- tabr\_options, 55
- tabrSyntax, 54
- tally\_notes (note-metadata), 33
- tally\_octaves (note-metadata), 33
- tally\_pitches (note-metadata), 33
- tie, 56
- time\_format (note-metadata), 33
- tp (transpose), 59
- track, 49, 56
- trackbind, 57, 58
- transpose, 10, 57, 59
- triplet (tuplet), 60
- tuning\_intervals (interval-helpers), 20
- tunings, 18, 57, 60
- tuplet, 60
- untie (tie), 56
- valid-notes, 61
- volta (repeats), 43
- x5 (chords), 7
- x7 (chords), 7
- x7s11 (chords), 7
- x7s5 (chords), 7
- x7s9 (chords), 7
- x9 (chords), 7
- x\_11 (chords), 7
- x\_13 (chords), 7
- xadd9 (chords), 7
- xaug (chords), 7
- xdim (chords), 7
- xdim7 (chords), 7
- xM (chords), 7
- xm (chords), 7
- xM11 (chords), 7
- xm11 (chords), 7
- xM13 (chords), 7
- xm13 (chords), 7
- xM6 (chords), 7
- xm6 (chords), 7
- xM7 (chords), 7
- xm7 (chords), 7
- xm7b5 (chords), 7
- xM7s11 (chords), 7
- xM9 (chords), 7
- xm9 (chords), 7
- xma9 (chords), 7
- xs2 (chords), 7
- xs4 (chords), 7