Package ‘systemfonts’

April 4, 2025

Type Package

Title System Native Font Finding

Version 1.2.2

Description Provides system native access to the font catalogue. As font handling varies between systems it is difficult to correctly locate installed fonts across different operating systems. The 'systemfonts' package provides bindings to the native libraries on Windows, macOS and Linux for finding font files that can then be used further by e.g. graphic devices. The main use is intended to be from compiled code but 'systemfonts' also provides access from R.

License MIT + file LICENSE

URL `https://github.com/r-lib/systemfonts`, `https://systemfonts.r-lib.org`

BugReports `https://github.com/r-lib/systemfonts/issues`

Depends R (>= 3.2.0)

Suggests covr, farver, graphics, knitr, rmarkdown, testthat (>= 2.1.0)

LinkingTo cpp11 (>= 0.2.1)

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.3.2

SystemRequirements fontconfig, freetype2

Config/Needs/website tidyverse/tidytemplate

Imports grid, jsonlite, lifecycle, tools, utils

Config/build/compilation-database true

NeedsCompilation yes

Author Thomas Lin Pedersen [aut, cre] (<`https://orcid.org/0000-0002-5147-4711`>), Jeroen Ooms [aut] (<`https://orcid.org/0000-0002-4035-0289`>), Devon Govett [aut] (Author of font-manager), Posit, PBC [cph, fnd]

Maintainer Thomas Lin Pedersen <`thomas.pedersen@posit.co`>

Repository CRAN

Date/Publication 2025-04-04 13:00:02 UTC

Contents

`add_fonts` *Add local font files to the search path*

Description

systemfonts is mainly about getting system native access to the fonts installed on the OS you are executing the code on. However, you may want to access fonts without doing a full installation, either because you want your project to be reproducible on all systems, because you don't have administrator priviliges on the system, or for a different reason entirely. `add_fonts()` provide a way to side load font files so that they are found during font matching. The function differs from `register_font()` and `register_variant()` in that they add the font file as-is using the family name etc that are provided by the font. `scan_local_fonts()` is run when systemfonts is loaded and will automatically add font files stored in `./fonts` (project local) and `~/fonts` (user local).

Usage

```
add_fonts(files)

scan_local_fonts()

clear_local_fonts()
```

Arguments

`files`	A character vector of font file paths or urls to add

Value

This function is called for its sideeffects

Font matching

During font matching, systemfonts has to look in three different locations. The font registry (populated by `register_font()`/`register_variant()`), the local fonts (populated with `add_fonts()`/`scan_local_fonts()`), and the fonts installed on the system. It does so in that order: registry > local > installed.

The matching performed at each step also differs. The fonts in the registry is only matched by family name. The local fonts are matched based on all the provided parameters (family, weight, italic, etc) in a way that is local to systemfonts, but try to emulate the system native matching. The installed fonts are matched using the system native matching functionality on macOS and Linux. On Windows the installed fonts are read from the system registry and matched using the same approach as for local fonts. Matching will always find a font no matter what you throw at it, defaulting to "sans" if nothing else is found.

Examples

```
# example code
empty_font <- system.file("unfont.ttf", package = "systemfonts")

add_fonts(empty_font)

clear_local_fonts()
```

`font_fallback` *Get the fallback font for a given string*

Description

A fallback font is a font to use as a substitute if the chosen font does not contain the requested characters. Using font fallbacks means that the user doesn't have to worry about mixing characters from different scripts or mixing text and emojies. Fallback is calculated for the full string and the result is platform specific. If no font covers all the characters in the string an undefined "best match" is returned. The best approach is to figure out which characters are not covered by your chosen font and figure out fallbacks for these, rather than just request a fallback for the full string.

Usage

```
font_fallback(
  string,
  family = "",
  italic = FALSE,
  bold = FALSE,
  path = NULL,
  index = 0
)
```

Arguments

`string`	The strings to find fallbacks for
`family`	The name of the font families to match
`italic`	logical indicating the font slant
`bold`	logical indicating whether the font weight
`path, index`	path an index of a font file to circumvent lookup based on family and style

Value

A data frame with a `path` and `index` column giving fallback for the specified string and font combinations

Examples

```
font_fallback("\U0001f604") # Smile emoji
```

`font_feature` *Define OpenType font feature settings*

Description

This function encapsulates the specification of OpenType font features. Some specific features have named arguments, but all available features can be set by using its specific 4-letter tag For a list of the 4-letter tags available see e.g. the overview on Wikipedia.

Usage

```
font_feature(ligatures = NULL, letters = NULL, numbers = NULL, ...)
```

Arguments

`ligatures`	Settings related to ligatures. One or more types of ligatures to turn on (see details).
`letters`	Settings related to the appearance of single letters (as opposed to ligatures that substitutes multiple letters). See details for supported values.
`numbers`	Settings related to the appearance of numbers. See details for supported values.
`...`	key-value pairs with the key being the 4-letter tag and the value being the setting (usually `TRUE` to turn it on).

Details

OpenType features are defined by a 4-letter tag along with an integer value. Often that value is a simple `0` (off) or `1` (on), but some features support additional values, e.g. stylistic alternates (`salt`) where a font may provide multiple variants of a letter and the value will be used to chose which one to use.

Common features related to appearance may be given with a long form name to either the `ligatures`, `letters`, or `numbers` argument to avoid remembering the often arbitrary 4-letter tag. Providing a long form name is the same as setting the tag to `1` and can thus not be used to set tags to other values.

The possible long form names are given below with the tag in parenthesis:

Ligatures

- `standard` (*liga*): Turns on standard multiple letter substitution
- `historical` (*hlig*): Use obsolete historical ligatures
- `contextual` (*clig*): Apply secondary ligatures based on the character patterns surrounding the potential ligature
- `discretionary` (*dlig*): Use ornamental ligatures

Letters

- `swash` (*cswh*): Use contextual swashes (ornamental decorations)
- `alternates` (*calt*): Use alternate letter forms based on the sourrounding pattern
- `historical` (*hist*): Use obsolete historical forms of the letters
- `localized` (*locl*): Use alternate forms preferred by the script language
- `randomize` (*rand*): Use random variants of the letters (e.g. to mimick handwriting)
- `alt_annotation` (*nalt*): Use alternate annotations (e.g. circled digits)
- `stylistic` (*salt*): Use a stylistic alternative form of the letter
- `subscript` (*subs*): Set letter in subscript
- `superscript` (*sups*): Set letter in superscript
- `titling` (*titl*): Use letter forms well suited for large text and titles
- `small_caps` (*smcp*): Use small caps variants of the letters

Numbers

- `lining` (*lnum*): Use number variants that rest on the baseline
- `oldstyle` (*onum*): Use old style numbers that use descender and ascender for various numbers
- `proportional` (*pnum*): Let numbers take up width based on the visual width of the glyph
- `tabular` (*tnum*): Enforce all numbers to take up the same width
- `fractions` (*frac*): Convert numbers separated by `/` into a fraction glyph
- `fractions_alt` (*afrc*): Use alternate fraction form with a horizontal divider

Value

A `font_feature` object

Examples

```
font_feature(letters = "stylistic", numbers = c("lining", "tabular"))

# Use the tag directly to access additional stylistic variants
font_feature(numbers = c("lining", "tabular"), salt = 2)
```

`font_info` *Query font-specific information*

Description

Get general information about a font, relative to a given size. Size specific measures will be returned in pixel units. The function is vectorised to the length of the longest argument.

Usage

```
font_info(
  family = "",
  italic = FALSE,
  bold = FALSE,
  size = 12,
  res = 72,
  path = NULL,
  index = 0
)
```

Arguments

`family`	The name of the font families to match
`italic`	logical indicating the font slant
`bold`	logical indicating whether the font weight
`size`	The pointsize of the font to use for size related measures
`res`	The ppi of the size related mesures
`path, index`	path an index of a font file to circumvent lookup based on family and style

Value

A data.frame giving info on the requested font + size combinations. The data.frame will contain the following columns:

path The path to the font file

index The 0-based index of the font in the fontfile

family The family name of the font

style The style name of the font

italic A logical giving if the font is italic

bold A logical giving if the font is bold

monospace A logical giving if the font is monospace

weight A factor giving the weight of the font

width A factor giving the width of the font

kerning A logical giving if the font supports kerning

color A logical giving if the font has color glyphs

scalable A logical giving if the font is scalable

vertical A logical giving if the font is vertical

n_glyphs The number of glyphs in the font

n_sizes The number of predefined sizes in the font

n_charmaps The number of character mappings in the font file

bbox A bounding box large enough to contain any of the glyphs in the font

max_ascend The maximum ascend of the tallest glyph in the font

max_descent The maximum descend of the most descending glyph in the font

max_advance_width The maximum horizontal advance a glyph can make

max_advance_height The maximum vertical advance a glyph can make

lineheight The height of a single line of text in the font

underline_pos The position of a potential underlining segment

underline_size The width the the underline

Examples

```
font_info('serif')

# Avoid lookup if font file is already known
sans <- match_fonts('sans')
font_info(path = sans$path, index = sans$index)
```

`glyph_info` *Query glyph-specific information from fonts*

Description

This function allows you to extract information about the individual glyphs in a font, based on a specified size. All size related measures are in pixel-units. The function is vectorised to the length of the `glyphs` vector.

Usage

```
glyph_info(
  glyphs,
  family = "",
  italic = FALSE,
  bold = FALSE,
  size = 12,
  res = 72,
  path = NULL,
  index = 0
)
```

Arguments

`glyphs`	A vector of glyphs. Strings will be split into separate glyphs automatically
`family`	The name of the font families to match
`italic`	logical indicating the font slant
`bold`	logical indicating whether the font weight
`size`	The pointsize of the font to use for size related measures
`res`	The ppi of the size related mesures
`path, index`	path an index of a font file to circumvent lookup based on family and style

Value

A data.frame with information about each glyph, containing the following columns:

glyph The glyph as a character
index The index of the glyph in the font file
width The width of the glyph
height The height of the glyph
x_bearing The horizontal distance from the origin to the leftmost part of the glyph
y_bearing The vertical distance from the origin to the top part of the glyph
x_advance The horizontal distance to move the cursor after adding the glyph
y_advance The vertical distance to move the cursor after adding the glyph
bbox The tight bounding box surrounding the glyph

`glyph_outline` *Get the outline of glyphs*

Description

This function allows you to retrieve the outline of glyphs as polygon coordinates. The glyphs are given as indexes into a font file and not as characters allowing you to retrieve outlines for glyphs that doesn't have a character counterpoint. Glyphs that are given as bitmaps are ignored.

Usage

```
glyph_outline(
  glyph,
  path,
  index = 0,
  size = 12,
  tolerance = 0.2,
  verbose = FALSE
)
```

Arguments

`glyph`	The index of the glyph in the font file
`path`	The path to the font file encoding the glyph
`index`	The index of the font in the font file
`size`	The size of the font in big points (1/72 inch)
`tolerance`	The deviation tolerance for decomposing bezier curves of the glyph. Given in the same unit as size. Smaller values give more detailed polygons
`verbose`	Should font and glyph loading errors be reported as warnings

Value

A data frame giving the outlines of the glyphs provide in `glyph`. It contains the columns `glyph` pointing to the element in the input it relates to, `contour` enumerating the contours the glyph consists of, and `x` and `y` giving the coordinates in big points

Examples

```
# Get the shape of s in the default font
font <- font_info()
glyph <- glyph_info("s", path = font$path, index = font$index)

s <- glyph_outline(glyph$index, font$path, font$index, size = 150)

plot(s$x, s$y, type = 'l')
```

glyph_raster *Render glyphs to raster image*

Description

Not all glyphs are encoded as vector outlines (emojis often not). Even for fonts that provide an outline you might be interested in a raster version. This function gives you just that. It converts a glyph into an optimized raster object that can be plotted with e.g. `graphics::rasterImage()` or `grid::grid.raster()`. For convenience, you can also use `glyph_raster_grob()` for plotting the result.

Usage

```
glyph_raster(
  glyph,
  path,
  index = 0,
  size = 12,
  res = 300,
  col = "black",
  verbose = FALSE
)
```

Arguments

`glyph`	The index of the glyph in the font file
`path`	The path to the font file encoding the glyph
`index`	The index of the font in the font file
`size`	The size of the font in big points (1/72 inch)
`res`	The resolution to render the glyphs to
`col`	The color of the glyph assuming the glyph doesn't have a native coloring
`verbose`	Should font and glyph loading errors be reported as warnings

Value

A list of nativeRaster objects (or `NULL` if it failed to render a given glyph). The nativeRasters have additional attributes attached. `"size"` will give the size of the glyph in big points and `"offset"` will give the location of the top-left corner of the raster with respect to where it should be rendered.

Examples

```
font <- font_info()
glyph <- glyph_info("R", path = font$path, index = font$index)

R <- glyph_raster(glyph$index, font$path, font$index, size = 150)
```

```
plot.new()
plot.window(c(0,150), c(0, 150), asp = 1)
rasterImage(R[[1]], 0, 0, attr(R[[1]], "size")[2], attr(R[[1]], "size")[1])
```

glyph_raster_grob *Convert an extracted glyph raster to a grob*

Description

This is a convenience function that helps in creating rasterGrob with the correct settings for the glyph. It takes inot account the sizing and offset returned by `glyph_raster()` and allows you to only consider the baseline position of the glyph.

Usage

```
glyph_raster_grob(glyph, x, y, ..., default.units = "bigpts")
```

Arguments

`glyph`	The nativeRaster object returned as one of the elements by `glyph_raster()`
`x, y`	The baseline location of the glyph
`...`	Arguments passed on to `grid::rasterGrob` `image` Any R object that can be coerced to a raster object. `width` A numeric vector or unit object specifying width. `height` A numeric vector or unit object specifying height. `just` The justification of the rectangle relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`. For numeric values, 0 means left alignment and 1 means right alignment. `hjust` A numeric vector specifying horizontal justification. If specified, overrides the `just` setting. `vjust` A numeric vector specifying vertical justification. If specified, overrides the `just` setting. `name` A character identifier. `gp` An object of class `"gpar"`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings. `vp` A Grid viewport object (or NULL). `interpolate` A logical value indicating whether to linearly interpolate the image (the alternative is to use nearest-neighbour interpolation, which gives a more blocky result).
`default.units`	A string indicating the default units to use if `x`, `y`, `width`, or `height` are only given as numeric vectors.

Value

A rasterGrob object

Examples

```
font <- font_info()
glyph <- glyph_info("R", path = font$path, index = font$index)

R <- glyph_raster(glyph$index, font$path, font$index, size = 150)

grob <- glyph_raster_grob(R[[1]], 50, 50)

grid::grid.newpage()
# Mark the baseline location
grid::grid.points(50, 50, default.units = "bigpts")
# Draw the glyph
grid::grid.draw(grob)
```

`match_fonts` *Find a system font by name and style*

Description

This function locates the font file (and index) best matching a name and optional style. A font file will be returned even if a perfect match isn't found, but it is not necessarily similar to the requested family and it should not be relied on for font substitution. The aliases `"sans"`, `"serif"`, `"mono"`, `"symbol"`, and `"emoji"` match to their respective system defaults (`""` is equivalent to `"sans"`). `match_font()` has been deprecated in favour of `match_fonts()` which provides vectorisation, as well as querying for different weights (rather than just "normal" and "bold") as well as different widths.

Usage

```
match_fonts(family, italic = FALSE, weight = "normal", width = "undefined")

match_font(family, italic = FALSE, bold = FALSE)
```

Arguments

`family`	The name of the font families to match
`italic`	logical indicating the font slant
`weight`	The weight to query for, either in numbers (`0`, `100`, `200`, `300`, `400`, `500`, `600`, `700`, `800`, or `900`) or strings (`"undefined"`, `"thin"`, `"ultralight"`, `"light"`, `"normal"`, `"medium"`, `"semibold"`, `"bold"`, `"ultrabold"`, or `"heavy"`). `NA` will be interpreted as `"undefined"`/`0`

`width`	The width to query for either in numbers (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`) or strings (`"undefined"`, `"ultracondensed"`, `"extracondensed"`, `"condensed"`, `"semicondensed"`, `"normal"`, `"semiexpanded"`, `"expanded"`, `"extraexpanded"`, or `"ultraexpanded"`). `NA` will be interpreted as `"undefined"`/`0`
`bold`	logical indicating whether the font weight

Value

A list containing the paths locating the font files, the 0-based index of the font in the files and the features for the font in case a registered font was located.

Font matching

During font matching, systemfonts has to look in three different locations. The font registry (populated by `register_font()`/`register_variant()`), the local fonts (populated with `add_fonts()`/`scan_local_fonts()`), and the fonts installed on the system. It does so in that order: registry > local > installed.

The matching performed at each step also differs. The fonts in the registry is only matched by family name. The local fonts are matched based on all the provided parameters (family, weight, italic, etc) in a way that is local to systemfonts, but try to emulate the system native matching. The installed fonts are matched using the system native matching functionality on macOS and Linux. On Windows the installed fonts are read from the system registry and matched using the same approach as for local fonts. Matching will always find a font no matter what you throw at it, defaulting to "sans" if nothing else is found.

Examples

```
# Get the system default sans-serif font in italic
match_fonts('sans', italic = TRUE)

# Try to match it to a thin variant
match_fonts(c('sans', 'serif'), weight = "thin")
```

`register_font` *Register font collections as families*

Description

By design, systemfonts searches the fonts installed natively on the system. It is possible, however, to register other fonts from e.g. font packages or local font files, that will get searched before searching any installed fonts. You can always get an overview over all registered fonts with the `registry_fonts()` function that works as a registry focused analogue to `system_fonts()`. If you wish to clear out the registry, you can either restart the R session or call `clear_registry()`.

Usage

```
register_font(
  name,
  plain,
  bold = plain,
  italic = plain,
  bolditalic = plain,
  features = font_feature()
)

registry_fonts()

clear_registry()
```

Arguments

`name`	The name the collection will be known under (i.e. *family*)
`plain`, `bold`, `italic`, `bolditalic`	Fontfiles for the different faces of the collection. can either be a filepath or a list containing a filepath and an index (only for font files containing multiple fonts). If not given it will default to the `plain` specification.
`features`	A `font_feature` object describing the specific OpenType font features to turn on for the registered font.

Details

`register_font` also makes it possible to use system fonts with traits that is not covered by the graphic engine in R. In plotting operations it is only possible to specify a family name and whether or not the font should be bold and/or italic. There are numerous fonts that will never get matched to this, especially because bold is only one of many weights.

Apart from granting a way to use new varieties of fonts, font registration also allows you to override the default `sans`, `serif`, and `mono` mappings, simply by registering a collection to the relevant default name. As registered fonts are searched first it will take precedence over the default.

Value

`register_font()` and `clear_registry()` returns `NULL` invisibly. `registry_fonts()` returns a data table in the same style as `system_fonts()` though less detailed and not based on information in the font file.

Font matching

During font matching, systemfonts has to look in three different locations. The font registry (populated by `register_font()`/`register_variant()`), the local fonts (populated with `add_fonts()`/`scan_local_fonts()`), and the fonts installed on the system. It does so in that order: registry > local > installed.

The matching performed at each step also differs. The fonts in the registry is only matched by family name. The local fonts are matched based on all the provided parameters (family, weight, italic, etc) in a way that is local to systemfonts, but try to emulate the system native matching. The installed fonts are matched using the system native matching functionality on macOS and Linux. On Windows the installed fonts are read from the system registry and matched using the same approach as for local fonts. Matching will always find a font no matter what you throw at it, defaulting to "sans" if nothing else is found.

Examples

```
# Create a random font collection
fonts <- system_fonts()
plain <- sample(which(!fonts$italic & fonts$weight <= 'normal'), 1)
bold <- sample(which(!fonts$italic & fonts$weight > 'normal'), 1)
italic <- sample(which(fonts$italic & fonts$weight <= 'normal'), 1)
bolditalic <- sample(which(fonts$italic & fonts$weight > 'normal'), 1)
register_font(
  'random',
  plain = list(fonts$path[plain], fonts$index[plain]),
  bold = list(fonts$path[bold], fonts$index[bold]),
  italic = list(fonts$path[italic], fonts$index[italic]),
  bolditalic = list(fonts$path[bolditalic], fonts$index[bolditalic])
)

# Look at your creation
registry_fonts()

# Reset
clear_registry()
```

register_variant *Register a font as a variant as an existing one*

Description

This function is a wrapper around `register_font()` that allows you to easily create variants of existing system fonts, e.g. to target different weights and/or widths, or for attaching OpenType features to a font.

Usage

```
register_variant(
  name,
  family,
  weight = NULL,
  width = NULL,
  features = font_feature()
)
```

Arguments

`name`	The new family name the variant should respond to
`family`	The name of an existing font family that this is a variant of
`weight`	One or two of `"thin"`, `"ultralight"`, `"light"`, `"normal"`, `"medium"`, `"semibold"`, `"bold"`, `"ultrabold"`, or `"heavy"`. If one is given it sets the weight for the whole variant. If two is given the first one defines the plain weight and the second the bold weight. If `NULL` then the variants of the given family closest to `"normal"` and `"bold"` will be chosen.
`width`	One of `"ultracondensed"`, `"extracondensed"`, `"condensed"`, `"semicondensed"`, `"normal"`, `"semiexpanded"`, `"expanded"`, `"extraexpanded"`, or `"ultraexpanded"` giving the width of the variant. If `NULL` then the width closest to `"normal"` will be chosen.
`features`	A `font_feature` object describing the specific OpenType font features to turn on for the registered font variant.

Font matching

During font matching, systemfonts has to look in three different locations. The font registry (populated by `register_font()`/`register_variant()`), the local fonts (populated with `add_fonts()`/`scan_local_fonts()`), and the fonts installed on the system. It does so in that order: registry > local > installed.

The matching performed at each step also differs. The fonts in the registry is only matched by family name. The local fonts are matched based on all the provided parameters (family, weight, italic, etc) in a way that is local to systemfonts, but try to emulate the system native matching. The installed fonts are matched using the system native matching functionality on macOS and Linux. On Windows the installed fonts are read from the system registry and matched using the same approach as for local fonts. Matching will always find a font no matter what you throw at it, defaulting to "sans" if nothing else is found.

Examples

```
# Get the default "sans" family
sans <- match_fonts("sans")$path
sans <- system_fonts()$family[system_fonts()$path == sans][1]

# Register a variant of it:
register_variant(
  "sans_ligature",
  sans,
  features = font_feature(ligatures = "discretionary")
)

registry_fonts()

# clean up
clear_registry()
```

`require_font` *Ensure font availability in a script*

Description

When running a script on a different machine you are not always in control of which fonts are installed on the system and thus how graphics created by the script ends up looking. `require_font()` is a way to specify your font requirements for a script. It will look at the available fonts and if the required font family is not present it will attempt to fetch it from one of the given repositories (in the order given). If that fails, it will either throw an error or, if `fallback` is given, provide an alias for the fallback so it maps to the required font.

Usage

```
require_font(
  family,
  fallback = NULL,
  dir = tempdir(),
  repositories = c("Google Fonts", "Font Squirrel"),
  error = TRUE
)
```

Arguments

`family`	The font family to require
`fallback`	An available font to fall back to if `family` cannot be found or downloaded
`dir`	The location to put the font file downloaded from repositories
`repositories`	The repositories to search for the font in case it is not available on the system. They will be tried in the order given. Currently only `"Google Fonts"` and `"Font Squirrel"` is available.
`error`	Should the function throw an error if unsuccessful?

Value

Invisibly `TRUE` if the font is available or `FALSE` if not (this can only be returned if `error = FALSE`)

Examples

```
# Should always work
require_font("sans")
```

`reset_font_cache` *Reset the system font cache*

Description

Building the list of system fonts is time consuming and is therefore cached. This, in turn, means that changes to the system fonts (i.e. installing new fonts), will not propagate to systemfonts. The solution is to reset the cache, which will result in the next call to e.g. `match_fonts()` will trigger a rebuild of the cache.

Usage

```
reset_font_cache()
```

Examples

```
all_fonts <- system_fonts()

##-- Install a new font on the system --##

all_fonts_new <- system_fonts()

## all_fonts_new will be equal to all_fonts

reset_font_cache()

all_fonts_new <- system_fonts()

## all_fonts_new will now contain the new font
```

`search_web_fonts` *Search font repositories for a font based on family name*

Description

While it is often advisable to visit the webpage for a font repository when looking for a font, in order to see examples etc, `search_web_fonts()` provide a quick lookup based on family name in the repositories supported by systemfonts (currently Google Fonts and Font Squirrel). The lookup is based on fuzzy matching provided by `utils::adist()` and the matching parameters can be controlled through ...

Usage

```
search_web_fonts(family, n_max = 10, ...)
```

Arguments

`family`	The font family name to look for
`n_max`	The maximum number of matches to return
`...`	Arguments passed on to `utils::adist`

`costs` a numeric vector or list with names partially matching ‘`insertions`’, ‘`deletions`’ and ‘`substitutions`’ giving the respective costs for computing the Levenshtein distance, or `NULL` (default) indicating using unit cost for all three possible transformations.

`counts` a logical indicating whether to optionally return the transformation counts (numbers of insertions, deletions and substitutions) as the `"counts"` attribute of the return value.

`fixed` a logical. If `TRUE` (default), the `x` elements are used as string literals. Otherwise, they are taken as regular expressions and `partial = TRUE` is implied (corresponding to the approximate string distance used by `agrep` with `fixed = FALSE`).

`partial` a logical indicating whether the transformed `x` elements must exactly match the complete `y` elements, or only substrings of these. The latter corresponds to the approximate string distance used by `agrep` (by default).

`ignore.case` a logical. If `TRUE`, case is ignored for computing the distances.

`useBytes` a logical. If `TRUE` distance computations are done byte-by-byte rather than character-by-character.

Value

A data.frame with the columns `family`, giving the family name of the matched font, and `repository` giving the repository it was found in.

Examples

```
# Requires an internet connection

# search_web_fonts("Spectral")
```

`shape_string` *Calculate glyph positions for strings*

Description

Do basic text shaping of strings. This function will use freetype to calculate advances, doing kerning if possible. It will not perform any font substitution or ligature resolving and will thus be much in line with how the standard graphic devices does text shaping. Inputs are recycled to the length of `strings`.

Usage

```
shape_string(
  strings,
  id = NULL,
  family = "",
  italic = FALSE,
  bold = FALSE,
  size = 12,
  res = 72,
  lineheight = 1,
  align = "left",
  hjust = 0,
  vjust = 0,
  width = NA,
  tracking = 0,
  indent = 0,
  hanging = 0,
  space_before = 0,
  space_after = 0,
  path = NULL,
  index = 0
)
```

Arguments

`strings`	A character vector of strings to shape
`id`	A vector grouping the strings together. If strings share an id the shaping will continue between strings
`family`	The name of the font families to match
`italic`	logical indicating the font slant
`bold`	logical indicating whether the font weight
`size`	The pointsize of the font to use for size related measures
`res`	The ppi of the size related mesures
`lineheight`	A multiplier for the lineheight
`align`	Within text box alignment, either `'left'`, `'center'`, or `'right'`
`hjust`, `vjust`	The justification of the textbox surrounding the text
`width`	The requested with of the string in inches. Setting this to something other than `NA` will turn on word wrapping.
`tracking`	Tracking of the glyphs (space adjustment) measured in 1/1000 em.
`indent`	The indent of the first line in a paragraph measured in inches.
`hanging`	The indent of the remaining lines in a paragraph measured in inches.
`space_before`, `space_after`	The spacing above and below a paragraph, measured in points
`path`, `index`	path an index of a font file to circumvent lookup based on family and style

Value

A list with two element: `shape` contains the position of each glyph, relative to the origin in the enclosing textbox. `metrics` contain metrics about the full strings.

`shape` is a data.frame with the following columns:

glyph The glyph as a character

index The index of the glyph in the font file

metric_id The index of the string the glyph is part of (referencing a row in the `metrics` data.frame)

string_id The index of the string the glyph came from (referencing an element in the `strings` input)

x_offset The x offset in pixels from the origin of the textbox

y_offset The y offset in pixels from the origin of the textbox

x_mid The x offset in pixels to the middle of the glyph, measured from the origin of the glyph

`metrics` is a data.frame with the following columns:

string The text the string consist of

width The width of the string

height The height of the string

left_bearing The distance from the left edge of the textbox and the leftmost glyph

right_bearing The distance from the right edge of the textbox and the rightmost glyph

top_bearing The distance from the top edge of the textbox and the topmost glyph

bottom_bearing The distance from the bottom edge of the textbox and the bottommost glyph

left_border The position of the leftmost edge of the textbox related to the origin

top_border The position of the topmost edge of the textbox related to the origin

pen_x The horizontal position of the next glyph after the string

pen_y The vertical position of the next glyph after the string

Examples

```
string <- "This is a long string\nLook; It spans multiple lines\nand all"

# Shape with default settings
shape_string(string)

# Mix styles within the same string
string <- c(
  "This string will have\na ",
  "very large",
  " text style\nin the middle"
)

shape_string(string, id = c(1, 1, 1), size = c(12, 24, 12))
```

string_metrics_dev *Get string metrics as measured by the current device*

Description

This function is much like `string_widths_dev()` but also returns the ascent and descent of the string making it possible to construct a tight bounding box around the string.

Usage

```
string_metrics_dev(
  strings,
  family = "",
  face = 1,
  size = 12,
  cex = 1,
  unit = "cm"
)
```

Arguments

`strings`	A character vector of strings to measure
`family`	The font families to use. Will get recycled
`face`	The font faces to use. Will get recycled
`size`	The font size to use. Will get recycled
`cex`	The cex multiplier to use. Will get recycled
`unit`	The unit to return the width in. Either `"cm"`, `"inches"`, `"device"`, or `"relative"`

Value

A data.frame with `width`, `ascent`, and `descent` columns giving the metrics in the requested unit.

See Also

Other device metrics: `string_widths_dev()`

Examples

```
# Get the metrics as measured in cm (default)
string_metrics_dev(c('some text', 'a string with descenders'))
```

`string_width` *Calculate the width of a string, ignoring new-lines*

Description

This is a very simple alternative to `shape_string()` that simply calculates the width of strings without taking any newline into account. As such it is suitable to calculate the width of words or lines that has already been splitted by `\n`. Input is recycled to the length of `strings`.

Usage

```
string_width(
  strings,
  family = "",
  italic = FALSE,
  bold = FALSE,
  size = 12,
  res = 72,
  include_bearing = TRUE,
  path = NULL,
  index = 0
)
```

Arguments

`strings`	A character vector of strings
`family`	The name of the font families to match
`italic`	logical indicating the font slant
`bold`	logical indicating whether the font weight
`size`	The pointsize of the font to use for size related measures
`res`	The ppi of the size related mesures
`include_bearing`	Logical, should left and right bearing be included in the string width?
`path, index`	path an index of a font file to circumvent lookup based on family and style

Value

A numeric vector giving the width of the strings in pixels. Use the provided `res` value to convert it into absolute values.

Examples

```
strings <- c('A short string', 'A very very looong string')
string_width(strings)
```

string_widths_dev *Get string widths as measured by the current device*

Description

For certain composition tasks it is beneficial to get the width of a string as interpreted by the device that is going to plot it. grid provides this through construction of a `textGrob` and then converting the corresponding grob width to e.g. cm, but this comes with a huge overhead. `string_widths_dev()` provides direct, vectorised, access to the graphic device for as high performance as possible.

Usage

```
string_widths_dev(
  strings,
  family = "",
  face = 1,
  size = 12,
  cex = 1,
  unit = "cm"
)
```

Arguments

`strings`	A character vector of strings to measure
`family`	The font families to use. Will get recycled
`face`	The font faces to use. Will get recycled
`size`	The font size to use. Will get recycled
`cex`	The cex multiplier to use. Will get recycled
`unit`	The unit to return the width in. Either `"cm"`, `"inches"`, `"device"`, or `"relative"`

Value

A numeric vector with the width of each of the strings given in `strings` in the unit given in `unit`

See Also

Other device metrics: `string_metrics_dev()`

Examples

```
# Get the widths as measured in cm (default)
string_widths_dev(c('a string', 'an even longer string'))
```

`str_split_emoji` *Split a string into emoji and non-emoji glyph runs*

Description

In order to do correct text rendering, the font needed must be figured out. A common case is rendering of emojis within a string where the system emoji font is used rather than the requested font. This function will inspect the provided strings and split them up in runs that must be rendered with the emoji font, and the rest. Arguments are recycled to the length of the `string` vector.

Usage

```
str_split_emoji(
  string,
  family = "",
  italic = FALSE,
  bold = FALSE,
  path = NULL,
  index = 0
)
```

Arguments

`string`	A character vector of strings that should be splitted.
`family`	The name of the font families to match
`italic`	logical indicating the font slant
`bold`	logical indicating whether the font weight
`path, index`	path an index of a font file to circumvent lookup based on family and style

Value

A data.frame containing the following columns:

string The substring containing a consecutive run of glyphs

id The index into the original `string` vector that the substring is part of

emoji A logical vector giving if the substring is a run of emojis or not

Examples

```
emoji_string <- "This is a joke\U0001f642. It should be obvious from the smiley"
str_split_emoji(emoji_string)
```

system_fonts *List all fonts installed on your system*

Description

List all fonts installed on your system

Usage

```
system_fonts()
```

Value

A data frame with a row for each font and various information in each column

Examples

```
# See all monospace fonts
fonts <- system_fonts()
fonts[fonts$monospace, ]
```

web-fonts *Download and add web font*

Description

In order to use a font in R it must first be made available locally. These functions facilitate the download and registration of fonts from online repositories.

Usage

```
get_from_google_fonts(family, dir = "~/fonts", woff2 = FALSE)

get_from_font_squirrel(family, dir = "~/fonts")
```

Arguments

`family`	The font family to download (case insensitive)
`dir`	Where to download the font to. The default places it in your user local font folder so that the font will be available automatically in new R sessions. Set to `tempdir()` to only keep the font for the session.
`woff2`	Should the font be downloaded in the woff2 format (smaller and more optimized)? Defaults to FALSE as the format is not supported on all systems

Value

A logical invisibly indicating whether a font was found and downloaded or not

Index