

# Package: tidytransit (via r-universe)

October 19, 2024

**Type** Package

**Title** Read, Validate, Analyze, and Map GTFS Feeds

**Version** 1.7.0

**Description** Read General Transit Feed Specification (GTFS) zipfiles into a list of R dataframes. Perform validation of the data structure against the specification. Analyze the headways and frequencies at routes and stops. Create maps and perform spatial analysis on the routes and stops. Please see the GTFS documentation here for more detail: [<https://gtfs.org/>](https://gtfs.org/).

**License** GPL

**LazyData** TRUE

**Depends** R (>= 3.6.0)

**Imports** gtfsio (>= 1.2.0), dplyr (>= 1.1.1), data.table (>= 1.12.8), rlang, sf, jsonlite, hms, digest, geodist

**Suggests** testthat (>= 3.1.5), knitr, markdown, rmarkdown, ggplot2, scales, lubridate, leaflet

**RoxygenNote** 7.3.2

**URL** <https://github.com/r-transit/tidytransit>

**BugReports** <https://github.com/r-transit/tidytransit/issues>

**VignetteBuilder** knitr

**Encoding** UTF-8

**NeedsCompilation** no

**Author** Flavio Poletti [aut, cre], Daniel Herszenhut [aut] (<https://orcid.org/0000-0001-8066-1105>), Mark Padgham [aut], Tom Buckley [aut], Danton Noriega-Goodwin [aut], Angela Li [ctb], Elaine McVey [ctb], Charles Hans Thompson [ctb], Michael Sumner [ctb], Patrick Hausmann [ctb], Bob Rudis [ctb], James Lamb [ctb], Alexandra Kapp [ctb], Kearey Smith [ctb], Dave Vautin [ctb], Kyle Walker [ctb], Davis Vaughan [ctb], Ryan Rymarczyk [ctb], Kirill Müller [ctb]

**Maintainer** Flavio Poletti <flavio.poletti@hotmail.ch>

**Repository** CRAN

**Date/Publication** 2024-10-18 13:50:02 UTC

## Contents

as_tidygtfs . . . . .	3
cluster_stops . . . . .	3
empty_strings_to_na . . . . .	4
filter_feed_by_area . . . . .	5
filter_feed_by_date . . . . .	6
filter_feed_by_stops . . . . .	6
filter_feed_by_trips . . . . .	7
filter_stops . . . . .	8
filter_stop_times . . . . .	8
get_route_frequency . . . . .	9
get_route_geometry . . . . .	10
get_stop_frequency . . . . .	11
get_trip_geometry . . . . .	12
gtfs_as_sf . . . . .	12
gtfs_duke . . . . .	13
gtfs_transform . . . . .	13
interpolate_stop_times . . . . .	14
na_to_empty_strings . . . . .	14
plot.tidygtfs . . . . .	15
print.tidygtfs . . . . .	16
raptor . . . . .	16
read_gtfs . . . . .	18
route_type_names . . . . .	19
set_servicepattern . . . . .	20
sf_as_tbl . . . . .	20
shapes_as_sf . . . . .	21
stops_as_sf . . . . .	22
stop_distances . . . . .	22
stop_group_distances . . . . .	23
summary.tidygtfs . . . . .	24
travel_times . . . . .	25
validate_gtfs . . . . .	27
write_gtfs . . . . .	29

<b>Index</b>	<b>30</b>
--------------	-----------

---

as_tidygtfs	<i>Convert another gtfs like object to a tidygtfs object</i>
-------------	--

---

**Description**

Convert another gtfs like object to a tidygtfs object

**Usage**

```
as_tidygtfs(x, ...)
```

**Arguments**

x	gtfs object
...	ignored

**Value**

a tidygtfs object

---

cluster_stops	<i>Cluster nearby stops within a group</i>
---------------	--

---

**Description**

Finds clusters of stops for each unique value in group\_col (e.g. stop\_name). Can be used to find different groups of stops that share the same name but are located more than max\_dist apart. gtfs\_stops is assigned a new column (named cluster\_colname) which contains the group\_col value and the cluster number.

**Usage**

```
cluster_stops(  
  gtfs_stops,  
  max_dist = 300,  
  group_col = "stop_name",  
  cluster_colname = "stop_name_cluster"  
)
```

**Arguments**

gtfs_stops	Stops table of a gtfs object. It is also possible to pass a tidygtfs object to enable piping.
max_dist	Only stop groups that have a maximum distance among them above this threshold (in meters) are clustered.
group_col	Clusters for are calculated for each set of stops with the same value in this column (default: stop_name)
cluster_colname	Name of the new column name. Can be the same as group_col to overwrite.

**Details**

`stats::kmeans()` is used for clustering.

**Value**

Returns a stops table with an added cluster column. If `gtfs_stops` is a tidygtfs object, a modified tidygtfs object is return

**Examples**

```
library(dplyr)
nyc_path <- system.file("extdata", "nyc_subway.zip", package = "tidytransit")
nyc <- read_gtfs(nyc_path)
nyc <- cluster_stops(nyc)

# There are 6 stops with the name "86 St" that are far apart
stops_86_St = nyc$stops %>%
  filter(stop_name == "86 St")

table(stops_86_St$stop_name_cluster)

stops_86_St %>% select(stop_id, stop_name, parent_station, stop_name_cluster) %>% head()

library(ggplot2)
ggplot(stops_86_St) +
  geom_point(aes(stop_lon, stop_lat, color = stop_name_cluster))
```

---

empty\_strings\_to\_na    *Convert empty strings ("" to NA values in all gtfs tables*

---

**Description**

Convert empty strings ("" to NA values in all gtfs tables

**Usage**

```
empty_strings_to_na(gtfs_obj)
```

**Arguments**

gtfs\_obj            gtfs feed (tidygtfs object)

**Value**

a gtfs\_obj where all empty strings in tables have been replaced with NA

**See Also**

[na\\_to\\_empty\\_strings\(\)](#)

---

`filter_feed_by_area`    *Filter a gtfs feed so that it only contains trips that pass a given area*

---

**Description**

Only stop\_times, stops, routes, services (in calendar and calendar\_dates), shapes, frequencies and transfers belonging to one of those trips are kept.

**Usage**

```
filter_feed_by_area(gtfs_obj, area)
```

**Arguments**

gtfs\_obj            gtfs feed (tidygtfs object)  
area                all trips passing through this area are kept. Either a bounding box (numeric vector with xmin, ymin, xmax, ymax) or a sf object.

**Value**

tidygtfs object with filtered tables

**See Also**

[filter\\_feed\\_by\\_stops](#), [filter\\_feed\\_by\\_trips](#), [filter\\_feed\\_by\\_date](#)

---

filter\_feed\_by\_date     *Filter a gtfs feed so that it only contains trips running on a given date*

---

### Description

Only stop\_times, stops, routes, services (in calendar and calendar\_dates), shapes, frequencies and transfers belonging to one of those trips are kept.

### Usage

```
filter_feed_by_date(
  gtfs_obj,
  extract_date,
  min_departure_time,
  max_arrival_time
)
```

### Arguments

gtfs\_obj             gtfs feed (tidygtfs object)

extract\_date        date to extract trips from this day (Date or "YYYY-MM-DD" string)

min\_departure\_time  
                      (optional) The earliest departure time. Can be given as "HH:MM:SS", hms object or numeric value in seconds.

max\_arrival\_time  
                      (optional) The latest arrival time. Can be given as "HH:MM:SS", hms object or numeric value in seconds.

### Value

tidygtfs object with filtered tables

### See Also

[filter\\_stop\\_times](#), [filter\\_feed\\_by\\_trips](#), [filter\\_feed\\_by\\_trips](#), [filter\\_feed\\_by\\_date](#)

---

filter\_feed\_by\_stops     *Filter a gtfs feed so that it only contains trips that pass the given stops*

---

### Description

Only stop\_times, stops, routes, services (in calendar and calendar\_dates), shapes, frequencies and transfers belonging to one of those trips are kept.

**Usage**

```
filter_feed_by_stops(gtfs_obj, stop_ids = NULL, stop_names = NULL)
```

**Arguments**

gtfs_obj	gtfs feed (tidygtfs object)
stop_ids	vector with stop_ids. You can either provide stop_ids or stop_names
stop_names	vector with stop_names (will be converted to stop_ids)

**Value**

tidygtfs object with filtered tables

**Note**

The returned gtfs\_obj likely contains more than just the stops given (i.e. all stops that belong to a trip passing the initial stop).

**See Also**

[filter\\_feed\\_by\\_trips](#), [filter\\_feed\\_by\\_trips](#), [filter\\_feed\\_by\\_date](#)

---

`filter_feed_by_trips` *Filter a gtfs feed so that it only contains a given set of trips*

---

**Description**

Only stop\_times, stops, routes, services (in calendar and calendar\_dates), shapes, frequencies and transfers belonging to one of those trips are kept.

**Usage**

```
filter_feed_by_trips(gtfs_obj, trip_ids)
```

**Arguments**

gtfs_obj	gtfs feed (tidygtfs object)
trip_ids	vector with trip_ids

**Value**

tidygtfs object with filtered tables

**See Also**

[filter\\_feed\\_by\\_stops](#), [filter\\_feed\\_by\\_area](#), [filter\\_feed\\_by\\_date](#)

---

filter_stops	<i>Get a set of stops for a given set of service ids and route ids</i>
--------------	--

---

### Description

Get a set of stops for a given set of service ids and route ids

### Usage

```
filter_stops(gtfs_obj, service_ids, route_ids)
```

### Arguments

gtfs_obj	gtfs feed (tidygtfs object)
service_ids	the service for which to get stops
route_ids	the route_ids for which to get stops

### Value

stops table for a given service or route

### Examples

```
library(dplyr)
local_gtfs_path <- system.file("extdata", "nyc_subway.zip", package = "tidytransit")
nyc <- read_gtfs(local_gtfs_path)
select_service_id <- filter(nyc$calendar, monday==1) %>% pull(service_id)
select_route_id <- sample_n(nyc$routes, 1) %>% pull(route_id)
filtered_stops_df <- filter_stops(nyc, select_service_id, select_route_id)
```

---

filter_stop_times	<i>Filter a stop_times table for a given date and timespan.</i>
-------------------	---

---

### Description

Filter a stop\_times table for a given date and timespan.

### Usage

```
filter_stop_times(gtfs_obj, extract_date, min_departure_time, max_arrival_time)
```



**Arguments**

gtfs\_obj           gtfs feed (tidygtfs object)  
 extract\_date      date to extract trips from this day (Date or "YYYY-MM-DD" string)  
 min\_departure\_time  
                   (optional) The earliest departure time. Can be given as "HH:MM:SS", hms object or numeric value in seconds.  
 max\_arrival\_time  
                   (optional) The latest arrival time. Can be given as "HH:MM:SS", hms object or numeric value in seconds.

**Value**

Filtered stop\_times data.table for `travel_times()` and `raptor()`.

**Examples**

```
feed_path <- system.file("extdata", "routing.zip", package = "tidytransit")
g <- read_gtfs(feed_path)

# filter the sample feed
stop_times <- filter_stop_times(g, "2018-10-01", "06:00:00", "08:00:00")
```

---

get\_route\_frequency    *Get Route Frequency*

---

**Description**

Calculate the number of departures and mean headways for routes within a given timespan and for given service\_ids.

**Usage**

```
get_route_frequency(
  gtfs_obj,
  start_time = "06:00:00",
  end_time = "22:00:00",
  service_ids = NULL
)
```

**Arguments**

gtfs\_obj           gtfs feed (tidygtfs object)  
 start\_time        analysis start time, can be given as "HH:MM:SS", hms object or numeric value in seconds.  
 end\_time          analysis period end time, can be given as "HH:MM:SS", hms object or numeric value in seconds.  
 service\_ids       A set of service\_ids from the calendar dataframe identifying a particular service id. If not provided, the service\_id with the most departures is used.

**Value**

a dataframe of routes with variables or headway/frequency in seconds for a route within a given time frame

**Note**

Some GTFS feeds contain a frequency data frame already. Consider using this instead, as it will be more accurate than what tidytransit calculates.

**Examples**

```
data(gtfs_duke)
routes_frequency <- get_route_frequency(gtfs_duke)
x <- order(routes_frequency$median_headways)
head(routes_frequency[x,])
```

---

get\_route\_geometry      *Get all trip shapes for a given route and service*

---

**Description**

Get all trip shapes for a given route and service

**Usage**

```
get_route_geometry(gtfs_sf_obj, route_ids = NULL, service_ids = NULL)
```

**Arguments**

gtfs_sf_obj	tidytransit gtfs object with sf data frames
route_ids	routes to extract
service_ids	service_ids to extract

**Value**

an sf dataframe for gtfs routes with a row/linestring for each trip

**Examples**

```
data(gtfs_duke)
gtfs_duke_sf <- gtfs_as_sf(gtfs_duke)
routes_sf <- get_route_geometry(gtfs_duke_sf)
plot(routes_sf[c(1,1350),])
```

---

get_stop_frequency	<i>Get Stop Frequency</i>
--------------------	---------------------------

---

### Description

Calculate the number of departures and mean headways for all stops within a given timespan and for given service\_ids.

### Usage

```
get_stop_frequency(  
  gtfs_obj,  
  start_time = "06:00:00",  
  end_time = "22:00:00",  
  service_ids = NULL,  
  by_route = TRUE  
)
```

### Arguments

gtfs_obj	gtfs feed (tidygtfs object)
start_time	analysis start time, can be given as "HH:MM:SS", hms object or numeric value in seconds.
end_time	analysis period end time, can be given as "HH:MM:SS", hms object or numeric value in seconds.
service_ids	A set of service_ids from the calendar dataframe identifying a particular service id. If not provided, the service_id with the most departures is used.
by_route	Default TRUE, if FALSE then calculate headway for any line coming through the stop in the same direction on the same schedule.

### Value

dataframe of stops with the number of departures and the headway (departures divided by timespan) in seconds as columns

### Note

Some GTFS feeds contain a frequency data frame already. Consider using this instead, as it will be more accurate than what tidytransit calculates.

### Examples

```
data(gtfs_duke)  
stop_frequency <- get_stop_frequency(gtfs_duke)  
x <- order(stop_frequency$mean_headway)  
head(stop_frequency[x,])
```

---

`get_trip_geometry`      *Get all trip shapes for given trip ids*

---

### Description

Get all trip shapes for given trip ids

### Usage

```
get_trip_geometry(gtfs_sf_obj, trip_ids)
```

### Arguments

`gtfs_sf_obj`      tidytransit gtfs object with sf data frames  
`trip_ids`          `trip_ids` to extract shapes

### Value

an sf dataframe for gtfs routes with a row/linestring for each trip

### Examples

```
data(gtfs_duke)
gtfs_duke <- gtfs_as_sf(gtfs_duke)
trips_sf <- get_trip_geometry(gtfs_duke, c("t_726295_b_19493_tn_41", "t_726295_b_19493_tn_40"))
plot(trips_sf[1,"shape_id"])
```

---

`gtfs_as_sf`                      *Convert stops and shapes to Simple Features*

---

### Description

Stops are converted to POINT sf data frames. Shapes are converted to a LINESTRING data frame. Note that this function replaces stops and shapes tables in `gtfs_obj`.

### Usage

```
gtfs_as_sf(gtfs_obj, skip_shapes = FALSE, crs = NULL, quiet = TRUE)
```

### Arguments

`gtfs_obj`          gtfs feed (tidygtfs object, created by `read_gtfs()`)  
`skip_shapes`      if TRUE, shapes are not converted. Default FALSE.  
`crs`                optional coordinate reference system (used by `sf::st_transform`) to transform lon/lat coordinates of stops and shapes  
`quiet`             boolean whether to print status messages

**Value**

tidygtfs object with stops and shapes as sf dataframes

**See Also**

[sf\\_as\\_tbl](#), [stops\\_as\\_sf](#), [shapes\\_as\\_sf](#)

---

gtfs\_duke

*Example GTFS data*

---

**Description**

Data obtained from <https://data.trilliumtransit.com/gtfs/duke-nc-us/duke-nc-us.zip>.

**Usage**

```
gtfs_duke
```

**Format**

An object of class tidygtfs (inherits from gtfs) of length 25.

**See Also**

[read\\_gtfs\(\)](#)

---

gtfs\_transform

*Transform coordinates of a gtfs feed*

---

**Description**

Transform coordinates of a gtfs feed

**Usage**

```
gtfs_transform(gtfs_obj, crs)
```

**Arguments**

gtfs_obj	gtfs feed (tidygtfs object)
crs	target coordinate reference system, used by sf::st_transform

**Value**

tidygtfs object with transformed stops and shapes sf dataframes

gtfs object with transformed sf tables

---

interpolate\_stop\_times

*Interpolate missing stop\_times linearly*

---

### Description

Interpolate missing stop\_times linearly

### Usage

```
interpolate_stop_times(x, use_shape_dist = TRUE)
```

### Arguments

**x** tidygtfs object or stop\_times table

**use\_shape\_dist** If TRUE, use shape\_dist\_traveled column from the shapes table for time interpolation (if that column is available). If FALSE or shape\_dist\_traveled is missing, times are interpolated equally between stops.

### Value

tidygtfs or stop\_times with interpolated arrival and departure times

### Examples

```
## Not run:
data(gtfs_duke)
print(gtfs_duke$stop_times[1:5, 1:5])

gtfs_duke_2 = interpolate_stop_times(gtfs_duke)
print(gtfs_duke_2$stop_times[1:5, 1:5])

gtfs_duke_3 = interpolate_stop_times(gtfs_duke, FALSE)
print(gtfs_duke_3$stop_times[1:5, 1:5])

## End(Not run)
```

---

na\_to\_empty\_strings *Convert NA values to empty strings ("")*

---

### Description

Convert NA values to empty strings ("")

### Usage

```
na_to_empty_strings(gtfs_obj)
```

**Arguments**

gtfs\_obj           gtfs feed (tidygtfs object)

**Value**

a gtfs\_obj where all NA strings in tables have been replaced with ""

**See Also**

[empty\\_strings\\_to\\_na\(\)](#)

---

plot.tidygtfs	<i>Plot GTFS stops and trips</i>
---------------	----------------------------------

---

**Description**

Plot GTFS stops and trips

**Usage**

```
## S3 method for class 'tidygtfs'  
plot(x, ...)
```

**Arguments**

x                   a tidygtfs object as read by [read\\_gtfs\(\)](#)  
...                 ignored for tidygtfs

**Value**

plot

**Examples**

```
local_gtfs_path <- system.file("extdata",  
                               "nyc_subway.zip",  
                               package = "tidytransit")  
nyc <- read_gtfs(local_gtfs_path)  
plot(nyc)
```

---

<code>print.tidygtfs</code>	<i>Print a GTFS object</i>
-----------------------------	----------------------------

---

### Description

Prints a GTFS object suppressing the class attribute and hiding the `validation_result` attribute, created with `validate_gtfs()`.

### Usage

```
## S3 method for class 'tidygtfs'  
print(x, ...)
```

### Arguments

<code>x</code>	a tidygtfs object as read by <code>read_gtfs()</code>
<code>...</code>	Optional arguments ultimately passed to <code>format</code> .

### Value

The GTFS object that was printed, invisibly

### Examples

```
## Not run:  
path = system.file("extdata",  
  "nyc_subway.zip",  
  package = "tidytransit")  
  
g = read_gtfs(path)  
print(g)  
  
## End(Not run)
```

---

<code>raptor</code>	<i>Calculate travel times from one stop to all reachable stops</i>
---------------------	--

---

### Description

`raptor` finds the minimal travel time, earliest or latest arrival time for all stops in `stop_times` with journeys departing from `stop_ids` within `time_range`.



**Usage**

```
raptor(
  stop_times,
  transfers,
  stop_ids,
  arrival = FALSE,
  time_range = 3600,
  max_transfers = NULL,
  keep = "all"
)
```

**Arguments**

stop_times	A (prepared) stop_times table from a gtfs feed. Prepared means that all stop time rows before the desired journey departure time should be removed. The table should also only include departures happening on one day. Use <a href="#">filter_stop_times()</a> for easier preparation.
transfers	Transfers table from a gtfs feed. In general no preparation is needed. Can be omitted if stop_times has been prepared with <a href="#">filter_stop_times()</a> .
stop_ids	Character vector with stop_ids from where journeys should start (or end). It is recommended to only use stop_ids that are related to each other, like different platforms in a train station or bus stops that are reasonably close to each other.
arrival	If FALSE (default), all journeys <i>start</i> from stop_ids. If TRUE, all journeys <i>end</i> at stop_ids.
time_range	Either a range in seconds or a vector containing the minimal and maximal departure time (i.e. earliest and latest possible journey departure time) as seconds or "HH:MM:SS" character. If arrival is TRUE, time_range describes the time window when journeys should end at stop_ids.
max_transfers	Maximum number of transfers allowed, no limit (NULL) as default.
keep	One of c("all", "shortest", "earliest", "latest"). By default, all journeys between stop_ids are returned. With <i>shortest</i> only the journey with the shortest travel time is returned. With <i>earliest</i> the journey arriving at a stop the earliest is returned, <i>latest</i> works accordingly.

**Details**

With a modified **Round-Based Public Transit Routing Algorithm** (RAPTOR) using data.table, earliest arrival times for all stops are calculated. If two journeys arrive at the same time, the one with the later departure time and thus shorter travel time is kept. By default, all journeys departing within time\_range that arrive at a stop are returned in a table. If you want all journeys *arriving* at stop\_ids within the specified time range, set arrival to TRUE.

Journeys are defined by a "from" and "to" stop\_id, a departure, arrival and travel time. Note that exact journeys (with each intermediate stop and route ids for example) are *not* returned.

For most cases, stop\_times needs to be filtered, as it should only contain trips happening on a single day, see [filter\\_stop\\_times\(\)](#). The algorithm scans all trips until it exceeds max\_transfers or all trips in stop\_times have been visited.

**Value**

A data.table with journeys (departure, arrival and travel time) to/from all stop\_ids reachable by stop\_ids.

**See Also**

[travel\\_times\(\)](#) for an easier access to travel time calculations via stop\_names.

**Examples**

```
nyc_path <- system.file("extdata", "nyc_subway.zip", package = "tidytransit")
nyc <- read_gtfs(nyc_path)

# you can use initial walk times to different stops in walking distance (arbitrary example values)
stop_ids_harlem_st <- c("301", "301N", "301S")
stop_ids_155_st <- c("A11", "A11N", "A11S", "D12", "D12N", "D12S")
walk_times <- data.frame(stop_id = c(stop_ids_harlem_st, stop_ids_155_st),
                        walk_time = c(rep(600, 3), rep(410, 6)), stringsAsFactors = FALSE)

# Use journeys departing after 7 AM with arrival time before 11 AM on 26th of June
stop_times <- filter_stop_times(nyc, "2018-06-26", 7*3600, 9*3600)

# calculate all journeys departing from Harlem St or 155 St between 7:00 and 7:30
rprr <- raptor(stop_times, nyc$transfers, walk_times$stop_id, time_range = 1800,
              keep = "all")

# add walk times to travel times
rprr <- merge(rprr, walk_times, by.x = "from_stop_id", by.y = "stop_id")
rprr$travel_time_incl_walk <- rprr$travel_time + rprr$walk_time

# get minimal travel times (with walk times) for all stop_ids
library(data.table)
shortest_travel_times <- setDT(rprr)[order(travel_time_incl_walk)][, .SD[1], by = "to_stop_id"]
hist(shortest_travel_times$travel_time, breaks = seq(0, 2*60)*60)
```

---

read\_gtfs

*Read and validate GTFS files*


---

**Description**

Reads a GTFS feed from either a local .zip file or an URL and validates them against GTFS specifications.

**Usage**

```
read_gtfs(path, files = NULL, quiet = TRUE, ...)
```

**Arguments**

path	The path to a GTFS .zip file.
files	A character vector containing the text files to be validated against the GTFS specification without the file extension (txt or geojson). If NULL (the default), all existing files are read.
quiet	Whether to hide log messages and progress bars (defaults to TRUE).
...	Can be used to pass on arguments to <code>gtfsio::import_gtfs()</code> . The parameters <code>files</code> and <code>quiet</code> are passed on by default.

**Value**

A tidygtfs object: a list of tibbles in which each entry represents a GTFS text file. Additional tables are stored in the `. sublist`.

**See Also**

`validate_gtfs()`, `write_gtfs()`

**Examples**

```
## Not run:
local_gtfs_path <- system.file("extdata", "nyc_subway.zip", package = "tidytransit")
gtfs <- read_gtfs(local_gtfs_path)
summary(gtfs)

gtfs <- read_gtfs(local_gtfs_path, files = c("trips", "stop_times"))
names(gtfs)

## End(Not run)
```

---

route_type_names	<i>Dataframe of route type id's and the names of the types (e.g. "Bus")</i>
------------------	---

---

**Description**

Extended GTFS Route Types: <https://developers.google.com/transit/gtfs/reference/extended-route-types>

**Usage**

```
route_type_names
```

**Format**

A data frame with 136 rows and 2 variables:

**route\_type** the id of route type

**route\_type\_name** name of the gtfs route type

**Source**

<https://gist.github.com/derhuerst/b0243339e22c310bee2386388151e11e>

---

set\_servicepattern      *Calculate service pattern ids for a GTFS feed*

---

**Description**

Each trip has a defined number of dates it runs on. This set of dates is called a service pattern in tidytransit. Trips with the same servicepattern id run on the same dates. In general, service\_id can work this way but it is not enforced by the GTFS standard.

**Usage**

```
set_servicepattern(
  gtfs_obj,
  id_prefix = "s_",
  hash_algo = "md5",
  hash_length = 7
)
```

**Arguments**

gtfs_obj	gtfs feed (tidygtfs object)
id_prefix	all servicepattern ids will start with this string
hash_algo	hashing algorithm used by digest
hash_length	length the hash should be cut to with substr(). Use -1 if the full hash should be used

**Value**

modified gtfs\_obj with added servicepattern list and a table linking trips and pattern (trip\_servicepatterns), added to gtfs\_obj\$. sublist.

---

sf\_as\_tbl      *Convert stops and shapes from sf objects to tibbles*

---

**Description**

Coordinates are transformed to lon/lat columns (stop\_lon/stop\_lat or shape\_pt\_lon/shape\_pt\_lat)

**Usage**

```
sf_as_tbl(gtfs_obj)
```

**Arguments**

gtfs\_obj           gtfs feed (tidygtfs object)

**Value**

tidygtfs object with stops and shapes converted to tibbles

**See Also**

[gtfs\\_as\\_sf](#)

---

shapes\_as\_sf           *Convert shapes into Simple Features Linestrings*

---

**Description**

Convert shapes into Simple Features Linestrings

**Usage**

```
shapes_as_sf(gtfs_shapes, crs = NULL)
```

**Arguments**

gtfs\_shapes       a gtfs\$shapes dataframe  
crs               optional coordinate reference system (used by sf::st\_transform) to transform lon/lat coordinates

**Value**

an sf dataframe for gtfs shapes

**See Also**

[gtfs\\_as\\_sf](#)

---

stops_as_sf	<i>Convert stops into Simple Features Points</i>
-------------	--

---

**Description**

Convert stops into Simple Features Points

**Usage**

```
stops_as_sf(stops, crs = NULL)
```

**Arguments**

stops	a gtfs\$stops dataframe
crs	optional coordinate reference system (used by sf::st_transform) to transform lon/lat coordinates

**Value**

an sf dataframe for gtfs routes with a point column

**See Also**

[gtfs\\_as\\_sf](#)

**Examples**

```
data(gtfs_duke)
some_stops <- gtfs_duke$stops[sample(nrow(gtfs_duke$stops), 40),]
some_stops_sf <- stops_as_sf(some_stops)
plot(some_stops_sf[, "stop_name"])
```

---

stop_distances	<i>Calculate distances between a given set of stops</i>
----------------	---

---

**Description**

Calculate distances between a given set of stops

**Usage**

```
stop_distances(gtfs_stops)
```

**Arguments**

gtfs_stops	gtfs stops table either as data frame (with at least stop_id, stop_lon and stop_lat columns) or as sf object.
------------	---

**Value**

Returns a data.frame with each row containing a pair of stop\_ids (columns from\_stop\_id and to\_stop\_id) and the distance between them (in meters)

**Note**

The resulting data.frame has  $nrow(gtfs\_stops)^2$  rows, distances calculations among all stops for large feeds should be avoided.

**Examples**

```
## Not run:
library(dplyr)

nyc_path <- system.file("extdata", "nyc_subway.zip", package = "tidytransit")
nyc <- read_gtfs(nyc_path)

nyc$stops %>%
  filter(stop_name == "Borough Hall") %>%
  stop_distances() %>%
  arrange(desc(distance))

#> # A tibble: 36 × 3
#>   from_stop_id to_stop_id distance
#>   <chr>        <chr>      <dbl>
#> 1 423          232         91.5
#> 2 423N        232         91.5
#> 3 423S        232         91.5
#> 4 423         232N        91.5
#> 5 423N        232N        91.5
#> 6 423S        232N        91.5
#> 7 423         232S        91.5
#> 8 423N        232S        91.5
#> 9 423S        232S        91.5
#> 10 232         423         91.5
#> # ... with 26 more rows

## End(Not run)
```

---

stop\_group\_distances *Calculates distances among stop within the same group column*

---

**Description**

By default calculates distances among stop\_ids with the same stop\_name.

**Usage**

```
stop_group_distances(gtfs_stops, by = "stop_name")
```

**Arguments**

`gtfs_stops` gtfs stops table either as data frame (with at least `stop_id`, `stop_lon` and `stop_lat` columns) or as `sf` object.

`by` group column, default: "stop\_name"

**Value**

data.frame with one row per group containing a distance matrix (`distances`), number of stop ids within that group (`n_stop_ids`) and distance summary values (`dist_mean`, `dist_median` and `dist_max`).

**Examples**

```
## Not run:
library(dplyr)

nyc_path <- system.file("extdata", "nyc_subway.zip", package = "tidytransit")
nyc <- read_gtfs(nyc_path)

stop_group_distances(nyc$stops)
#> # A tibble: 380 × 6
#>   stop_name distances      n_stop_ids dist_mean dist_median dist_max
#>   <chr>      <list>          <dbl>    <dbl>    <dbl>    <dbl>
#> 1 86 St      <dbl [18 × 18]>      18    5395.    5395.   21811.
#> 2 79 St      <dbl [6 × 6]>         6   19053.   19053.   19053.
#> 3 Prospect Av <dbl [6 × 6]>         6   18804.   18804.   18804.
#> 4 77 St      <dbl [6 × 6]>         6   16947.   16947.   16947.
#> 5 59 St      <dbl [6 × 6]>         6   14130.   14130.   14130.
#> 6 50 St      <dbl [9 × 9]>         9    7097.    7097.   14068.
#> 7 36 St      <dbl [6 × 6]>         6   12496.   12496.   12496.
#> 8 8 Av       <dbl [6 × 6]>         6   11682.   11682.   11682.
#> 9 7 Av       <dbl [9 × 9]>         9    5479.    5479.   10753.
#> 10 111 St    <dbl [9 × 9]>         9    3877.    3877.    7753.
#> # ... with 370 more rows

## End(Not run)
```

---

summary.tidygtfs

*GTFS feed summary*


---

**Description**

GTFS feed summary

**Usage**

```
## S3 method for class 'tidygtfs'
summary(object, ...)
```



**Arguments**

object            a tidygtfs object as read by `read_gtfs()`  
 ...                ignored for tidygtfs

**Value**

the tidygtfs object, invisibly

---

travel_times	<i>Calculate shortest travel times from a stop to all reachable stops</i>
--------------	---

---

**Description**

Function to calculate the shortest travel times from a stop (given by `stop_name`) to all other `stop_names` of a feed. `filtered_stop_times` needs to be created before with `filter_stop_times()` or `filter_feed_by_date()`.

**Usage**

```
travel_times(
  filtered_stop_times,
  stop_name,
  time_range = 3600,
  arrival = FALSE,
  max_transfers = NULL,
  max_departure_time = NULL,
  return_coords = FALSE,
  return_DT = FALSE,
  stop_dist_check = 300
)
```

**Arguments**

`filtered_stop_times`    `stop_times` data.table (with transfers and stops tables as attributes) created with `filter_stop_times()` where the departure or arrival time has been set.

`stop_name`               Stop name for which travel times should be calculated. A vector with multiple names can be used.

`time_range`             Either a range in seconds or a vector containing the minimal and maximal departure time (i.e. earliest and latest possible journey departure time) as seconds or "HH:MM:SS" character. If `arrival` is TRUE, `time_range` describes the time window when journeys should end at `stop_name`.

`arrival`                 If FALSE (default), all journeys *start* from `stop_name`. If TRUE, all journeys *end* at `stop_name`.

`max_transfers`         The maximum number of transfers. No limit if NULL

max_departure_time	Deprecated. Use <code>time_range</code> to set the latest possible departure time.
return_coords	Returns stop coordinates (lon/lat) as columns. Default is FALSE.
return_DT	<code>travel_times()</code> returns a data.table if TRUE. Default is FALSE which returns a tibble/tbl_df.
stop_dist_check	<code>stop_names</code> are not structured identifiers like <code>stop_ids</code> or <code>parent_stations</code> , so it's possible that stops with the same name are far apart. <code>travel_times()</code> errors if the distance among <code>stop_ids</code> with the same name is above this threshold (in meters). Use FALSE to turn check off. However, it is recommended to either use <code>raptor()</code> or fix the feed (see <code>cluster_stops()</code> ) in case of warnings.

### Details

This function allows easier access to `raptor()` by using stop names instead of ids and returning shortest travel times by default.

Note however that `stop_name` might not be a suitable identifier for a feed. It is possible that multiple stops have the same name while not being related or geographically close to each other. `stop_group_distances()` and `cluster_stops()` can help identify and fix issues with `stop_names`.

### Value

A table with travel times to/from all stops reachable by `stop_name` and their corresponding journey departure and arrival times.

### Examples

```
library(dplyr)

# 1) Calculate travel times from two closely related stops
# The example dataset gtfs_duke has missing times (allowed in gtfs) which is
# why we run interpolate_stop_times beforehand
gtfs = interpolate_stop_times(gtfs_duke)

tts1 = gtfs %>%
  filter_feed_by_date("2019-08-26") %>%
  travel_times(c("Campus Dr at Arts Annex (WB)", "Campus Dr at Arts Annex (EB)"),
              time_range = c("14:00:00", "15:30:00"))

# you can use either filter_feed_by_date or filter_stop_times to prepare the feed
# the result is the same
tts2 = gtfs %>%
  filter_stop_times("2019-08-26", "14:00:00") %>%
  travel_times(c("Campus Dr at Arts Annex (WB)", "Campus Dr at Arts Annex (EB)"),
              time_range = 1.5*3600) # 1.5h after 14:00

all(tts1 == tts2)
# It's recommended to store the filtered feed, since it can be time consuming to
# run it for every travel time calculation, see the next example steps
```

```

# 2) separate filtering and travel time calculation for a more granular analysis
# stop_names in this feed are not restricted to an area, create clusters of stops to fix
nyc_path <- system.file("extdata", "nyc_subway.zip", package = "tidytransit")
nyc <- read_gtfs(nyc_path)
nyc <- cluster_stops(nyc, group_col = "stop_name", cluster_colname = "stop_name")

# Use journeys departing after 7 AM with arrival time before 9 AM on 26th June
stop_times <- filter_stop_times(nyc, "2018-06-26", 7*3600, 9*3600)

# Calculate travel times from "34 St - Herald Sq"
tts <- travel_times(stop_times, "34 St - Herald Sq", return_coords = TRUE)

# only keep journeys under one hour for plotting
tts <- tts %>% filter(travel_time <= 3600)

# travel time to Queensboro Plaza is 810 seconds, 13:30 minutes
tts %>%
  filter(to_stop_name == "Queensboro Plaza") %>%
  mutate(travel_time = hms::hms(travel_time))

# plot a simple map showing travel times to all reachable stops
# this can be expanded to isochron maps
library(ggplot2)
ggplot(tts) + geom_point(aes(x=to_stop_lon, y=to_stop_lat, color = travel_time))

```

---

 validate\_gtfs

*Validate GTFS feed*


---

## Description

Validates the GTFS object against GTFS specifications and raises warnings if required files/fields are not found. This function is called in [read\\_gtfs\(\)](#).

## Usage

```
validate_gtfs(gtfs_obj, files = NULL, warnings = TRUE)
```

## Arguments

gtfs_obj	gtfs object (i.e. a list of tables, not necessary a tidygtfs object)
files	A character vector containing the text files to be validated against the GTFS specification without the file extension (txt or geojson). If NULL (the default), the provided GTFS feed is validated against all possible GTFS text files.
warnings	Whether to display warning messages (defaults to TRUE).

## Details

Note that this function just checks if required files or fields are missing. There's no validation for internal consistency (e.g. no departure times before arrival times or calendar covering a reasonable period).

## Value

A `validation_result` tibble containing the validation summary of all possible fields from the specified files.

## Details

GTFS object's files and fields are validated against the GTFS specifications as documented in [GTFS Schedule Reference](#):

- GTFS feeds are considered valid if they include all required files and fields. If a required file/field is missing the function (optionally) raises a warning.
- Optional files/fields are listed in the reference above but are not required, thus no warning is raised if they are missing.
- Extra files/fields are those who are not listed in the reference above (either because they refer to a specific GTFS extension or due to any other reason).

Note that some files (`calendar.txt`, `calendar_dates.txt` and `feed_info.txt`) are conditionally required. This means that:

- `calendar.txt` is initially set as a required file. If it's not present, however, it becomes optional and `calendar_dates.txt` (originally set as optional) becomes required.
- `feed_info.txt` is initially set as an optional file. If `translations.txt` is present, however, it becomes required.

## Examples

```
validate_gtfs(gtfs_duke)

## Not run:
local_gtfs_path <- system.file("extdata", "nyc_subway.zip", package = "tidytransit")
gtfs <- read_gtfs(local_gtfs_path)
attr(gtfs, "validation_result")

gtfs$shapes <- NULL
validation_result <- validate_gtfs(gtfs)

# should raise a warning
gtfs$stop_times <- NULL
validation_result <- validate_gtfs(gtfs)

## End(Not run)
```

---

write_gtfs	<i>Write a tidygtfs object to a zip file</i>
------------	--

---

**Description**

Write a tidygtfs object to a zip file

**Usage**

```
write_gtfs(gtfs_obj, zipfile, compression_level = 9, as_dir = FALSE)
```

**Arguments**

gtfs_obj	gtfs feed (tidygtfs object)
zipfile	path to the zip file the feed should be written to. The file is overwritten if it already exists.
compression_level	a number between 1 and 9, defaults to 9 (best compression).
as_dir	if TRUE, the feed is not zipped and zipfile is used as a directory path. The directory will be overwritten if it already exists.

**Value**

Invisibly returns gtfs\_obj

**Note**

Auxiliary tidytransit tables (e.g. dates\_services) are not exported. Calls `gtfsio::export_gtfs()` after preparing the data.

**See Also**

[read\\_gtfs\(\)](#)

# Index

## \* datasets

gtfs\_duke, 13  
route\_type\_names, 19

as\_tidygtfs, 3

cluster\_stops, 3  
cluster\_stops(), 26

empty\_strings\_to\_na, 4  
empty\_strings\_to\_na(), 15

filter\_feed\_by\_area, 5, 7  
filter\_feed\_by\_date, 5, 6, 6, 7  
filter\_feed\_by\_date(), 25  
filter\_feed\_by\_stops, 5, 6, 7  
filter\_feed\_by\_trips, 5–7, 7  
filter\_stop\_times, 6, 8  
filter\_stop\_times(), 17, 25  
filter\_stops, 8

get\_route\_frequency, 9  
get\_route\_geometry, 10  
get\_stop\_frequency, 11  
get\_trip\_geometry, 12  
gtfs\_as\_sf, 12, 21, 22  
gtfs\_duke, 13  
gtfs\_transform, 13  
gtfsio::export\_gtfs(), 29  
gtfsio::import\_gtfs(), 19

interpolate\_stop\_times, 14

na\_to\_empty\_strings, 14  
na\_to\_empty\_strings(), 5

plot.tidygtfs, 15  
print.tidygtfs, 16

raptor, 16  
raptor(), 9, 26

read\_gtfs, 18  
read\_gtfs(), 12, 13, 15, 16, 25, 27, 29  
route\_type\_names, 19

set\_servicepattern, 20  
sf\_as\_tbl, 13, 20  
shapes\_as\_sf, 13, 21  
stats::kmeans(), 4  
stop\_distances, 22  
stop\_group\_distances, 23  
stop\_group\_distances(), 26  
stops\_as\_sf, 13, 22  
summary.tidygtfs, 24

travel\_times, 25  
travel\_times(), 9, 18

validate\_gtfs, 27  
validate\_gtfs(), 16, 19

write\_gtfs, 29  
write\_gtfs(), 19