

Package: addr (via r-universe)

October 8, 2026

Title Clean, Parse, Harmonize, Match, and Geocode Messy Real-World US Addresses

Version 2.0.0

Description Cleans, parses, standardizes, matches, and geocodes messy, real-world US addresses. Uses the included usaddress library to tag address components and build addr vector objects composed of addr_part vectors for number, street, and place. These vectors can be standardized, matched, joined, and used as data-frame columns, allowing standard R tools to work with nested address structures.

URL <https://github.com/geomarker-io/addr/>, <https://geomarker.io/addr/>

BugReports <https://github.com/geomarker-io/addr/issues/>

License MIT + file LICENSE

Copyright See file inst/COPYRIGHTS for vendored Rust crate copyright information.

Encoding UTF-8

Roxygen list(markdown = TRUE)

Suggests arrow, dplyr, leaflet, mirai, pkgload, sf, testthat (>= 3.0.0), wk, withr

Imports digest, nanoparquet, s2, S7, stow (>= 0.3.0), stringdist, tibble, vctrs

Remotes cole-brokamp/stow

SystemRequirements Cargo (>= 1.71), rustc (>= 1.71)

Config/testthat/edition 3

Depends R (>= 4.2.0)

Config/rextendr/version 0.4.2

Collate 'addr_helpers.R' 'addr_part.R' 'addr.R'
'addr_fuzzy_left_join.R' 'addr_left_join.R' 'addr_match.R'
'addr_part_match.R' 'addr_tag_maps.R' 'addr_vectors.R'
'as_addr.R' 'clean.R' 'county_fips.R' 'data.R'

'extendr-wrappers.R' 'fuzzy_match.R' 'geocode.R' 'nad.R'
 'phonetic_street_key.R' 'place_zip.R' 'taf.R' 'tag_usaddress.R'
 'tiger.R' 'zzz.R'

Config/roxygen2/version 8.0.0

Config/pak/sysreqs libabsl-dev cmake libssl-dev libclang-dev

Repository <https://geomarker-io.r-universe.dev>

Date/Publication 2026-08-28 20:04:42 UTC

RemoteUrl <https://github.com/geomarker-io/addr>

RemoteRef v2.0.0

RemoteSha 2340946396d131a2bcd1d493961b1c372f6ce352

Contents

addr_fuzzy_left_join	3
addr_fuzzy_match	4
addr_left_join	6
addr_match	8
addr_match_stage	10
addr_number	11
as_addr	14
clean_address_text	15
county_fips_lookup	16
elh_data	17
fuzzy_match	17
geocode	18
geocode_stage	22
geocode_table	23
match_addr_number	23
match_addr_street	24
match_zipcodes	28
nad	29
nad_catalog	32
nad_dataset	32
nad_example_data	33
nad_manifest	34
phonetic_street_key	34
taf	35
taf_catalog	36
taf_dataset	37
taf_manifest	38
taf_needed_counties	39
tag_usaddress	40
tiger_addr_feat	42
tiger_feat_names	42
voter_addresses	43
zipcode_variant	44

addr_fuzzy_left_join *Left join two data frames using fuzzy addr matching*

Description

This wraps the addr fuzzy matching helpers and returns a left-join style result. The addr columns are matched by index and rows are expanded for one-to-many or many-to-many matches.

See `addr_match()` and `addr_left_join()` for a faster alternative that returns one selected match instead of all fuzzy matches.

Usage

```
addr_fuzzy_left_join(
  x,
  y,
  by = "addr",
  addr_fields = NULL,
  suffix = c(".x", ".y"),
  progress = interactive()
)
```

Arguments

<code>x, y</code>	data frames or tibbles with an addr column
<code>by</code>	addr column name in x (and y if the same); or a length-2 character vector of <code>c(x_col, y_col)</code>
<code>addr_fields</code>	a named vector of OSA maximum distances. Defaults are used for fields that are not supplied; see Details.
<code>suffix</code>	character vector of length 2 used to suffix duplicate columns
<code>progress</code>	logical; show progress bar while processing matched ZIP groups?

Details

`addr_fuzzy_left_join()` matches addresses within ZIP code groups, so maximum distances for place fields are ignored. Defaults for `addr_fields`:

- `number_prefix`: 0
- `number_digits`: 0
- `number_suffix`: 0
- `street_predirectional`: 0
- `street_premodifier`: 0
- `street_pretype`: 0
- `street_name`: 1
- `street_posttype`: 0
- `street_postdirectional`: 0

Value

a data frame with left-join semantics; note that *row order will be changed compared to x*

Examples

```
my_addr <-
  tibble::tibble(address = voter_addresses()[1:10],
                 addr = as_addr(address),
                 id = sprintf("id_%04d", seq_len(10)))
the_addr <- nad_example_data()
addr_fuzzy_left_join(my_addr, the_addr, c("addr", "nad_addr"))
```

addr_fuzzy_match	<i>Fuzzy match addr vectors using field-specific string distances</i>
------------------	---

Description

addr_fuzzy_match() matches two addr vectors using more than one address field.

fuzzy_match_addr_field() matches two addr vectors using a single address field.

Distances between address tags are defined using optimized string alignment; see fuzzy_match() and stringdist::stringdist() for more details.

Usage

```
addr_fuzzy_match(x, y, addr_fields = NULL)

fuzzy_match_addr_field(
  x,
  y,
  addr_field = c("number_prefix", "number_digits", "number_suffix",
                "street_predirectional", "street_premodifier", "street_pretype", "street_name",
                "street_posttype", "street_postdirectional", "place_name", "place_state",
                "place_zipcode"),
  osa_max_dist = 0
)
```

Arguments

x	addr vector to match
y	addr vector to match to
addr_fields	a named vector of OSA maximum distances. Defaults are used for fields that are not supplied; see Details.
addr_field	character name of single addr field to match on
osa_max_dist	maximum optimized string alignment distance used as threshold for matching on single addr field

Details

Defaults for `addr_fields`:

- `number_prefix`: 0
- `number_digits`: 0
- `number_suffix`: 0
- `street_predirectional`: 0
- `street_premodifier`: 0
- `street_pretype`: 0
- `street_name`: 1
- `street_posttype`: 0
- `street_postdirectional`: 0
- `place_name`: 0
- `place_state`: 0
- `place_zipcode`: 0

When fuzzy matching `street_name`, the "phonetic_street_key" prefilter is automatically used (see `?fuzzy_match`).

Value

a list of integer vectors representing the position of the best matching address(es) in `y` for each address in `x`

Examples

```
x_addr <- as_addr(c("123 Main St.", "333 Burnet Ave", "3333 Foofy Ave"))
y_addr <- as_addr(c("0000 Main Street", "3333 Burnet Avenue", "222 Burnet Ave"))

# no matches with defaults
addr_fuzzy_match(x_addr, y_addr)

# match on osa_max_dist of 2 for the address number
addr_fuzzy_match(x_addr, y_addr, addr_fields = c("number_digits" = 2))

# ignore address number when matching
addr_fuzzy_match(x_addr, y_addr, addr_fields = c("number_digits" = Inf))

fuzzy_match_addr_field(
  as_addr(c("123 Main St.", "3333 Burnet Ave", "3333 Foofy Ave")),
  as_addr(c("0000 Main Street", "0000 Burnet Avenue", "222 Burnet Ave")),
  addr_field = "street_name", osa_max_dist = 1
)

# empty address fields have an OSA distance of zero and always match
fuzzy_match_addr_field(
```

```

as_addr(c("123 Main St.", "3333 Burnet Ave", "3333 Foofy Ave")),
as_addr(c("0000 Main Street", "0000 Burnet Avenue", "222 Burnet Ave")),
addr_field = "number_prefix"
)

```

addr_left_join

Left join two data frames using addr matching

Description

addr_left_join() is a convenience wrapper around [addr_match\(\)](#) that returns a left-join style result. It expands rows of x for duplicate rows in the original y that share the exact matched addr, but it does not return multiple distinct candidate addresses from y. addr_match() still selects a single best address before this wrapper expands exact duplicates.

Usage

```

addr_left_join(
  x,
  y,
  by = "addr",
  suffix = c(".x", ".y"),
  zip_variants = TRUE,
  zip_variant = c("minus1", "plus1", "sub5", "sub4", "swap"),
  name_phonetic_dist = 2L,
  name_fuzzy_dist = 1L,
  number_fuzzy_dist = 1L,
  match_street_type = c("exact", "compatible", "ignore"),
  match_street_directional = c("exact", "swap", "ignore"),
  progress = interactive(),
  match_prepared = NULL
)

```

Arguments

x, y	data frames or tibbles with an addr column
by	addr column name in x (and y if the same); or a length-2 character vector of c(x_col, y_col)
suffix	character vector of length 2 used to suffix duplicate columns
zip_variants	logical; fuzzy match to common variants of x in y?
zip_variant	character vector; zipcode variant types to use when zip_variants is TRUE; see ?zipcode_variant
name_phonetic_dist	integer; maximum optimized string alignment distance between phonetic_street_key() of x and y to consider a possible match

name_fuzzy_dist
integer; maximum optimized string alignment distance between @name of x and y to consider a possible match

number_fuzzy_dist
integer; maximum optimized string alignment distance between addr_number strings in x and y to consider a possible match.

match_street_type
character; how to compare street pretype and posttype when selecting street candidates. "exact" requires pretype to match pretype and posttype to match posttype; "compatible" treats blank type fields as unknown but rejects candidates when known type information conflicts; "ignore" does not use street type fields when selecting candidates.

match_street_directional
character; how to compare street predirectional and postdirectional when selecting street candidates. "exact" requires predirectional to match predirectional and postdirectional to match postdirectional; "swap" also permits predirectional to match postdirectional and postdirectional to match predirectional; "ignore" does not use street directional fields when selecting candidates.

progress
logical; show addr_match() progress?

match_prepared
optional prepared addr_match_index for the y addr column, usually from addr_match_prepare(). When supplied, addr_left_join() validates that it is equivalent to the y addr column before reusing it for matching.

Value

A data frame with left-join semantics. Duplicate rows in y with the exact same matched addr are all returned. Partial ZIP-only or street-only matches do not expand to multiple candidate rows in y.

Examples

```
the_addr <- nad_example_data()
my_addr <- tibble::tibble(
  addr = as_addr(voter_addresses()[1:25]),
  id = 1:25
)

d <- addr_left_join(
  my_addr,
  the_addr,
  by = c("addr", "nad_addr"),
  match_prepared = nad_example_data(match_prepared = TRUE)
)

d

# some addresses may match with more than one address in NAD
# since matching does not consider subaddress (e.g. "line two")
# take the first row in these cases

table(addr_match_stage(d$nad_addr.y[!duplicated(d$id)]))
```

addr_match

*Match addr vectors***Description**

A single addr in y is chosen for each addr in x. Matching is staged to reduce the search space: ZIP codes are matched first, street names are then matched within each matched ZIP code, and street numbers are finally matched within each matched street and ZIP code combination. If more than one candidate addr remains in y after these stages, the first candidate in y is returned.

Missing or empty address components that cannot be matched at any stage are left missing in the returned addr() values. Rows with a matched ZIP code but no street match return an addr with only @place@zipcode filled; rows with matched ZIP code and street but no number match also return the matched @street.

addr_match() accepts raw reference data and prepares it internally, which is the right default for one-off matching jobs. addr_match_prepare() becomes useful when the same reference y will be reused across multiple calls to addr_match(), because it caches the deduplicated reference addresses and ZIP/street/number candidate lookups once instead of rebuilding them on every call.

Preparing y once avoids recomputing unique(y), ZIP-code groups, and exact street/number candidate lookups each time you call addr_match() with the same reference addresses. For a single end-to-end match, preparing y explicitly does not remove that work; it only moves it outside addr_match().

Usage

```
addr_match(
  x,
  y,
  zip_variants = TRUE,
  zip_variant = c("minus1", "plus1", "sub5", "sub4", "swap"),
  name_phonetic_dist = 2L,
  name_fuzzy_dist = 1L,
  number_fuzzy_dist = 1L,
  match_street_type = c("exact", "compatible", "ignore"),
  match_street_directional = c("exact", "swap", "ignore"),
  progress = interactive()
)

addr_match_prepare(y, progress = interactive())
```

Arguments

x	addr vector to match
y	addr vector to match against, or a prepared addr_match_index created by addr_match_prepare()
zip_variants	logical; fuzzy match to common variants of x in y?
zip_variant	character vector; zipcode variant types to use when zip_variants is TRUE; see ?zipcode_variant

name_phonetic_dist	integer; maximum optimized string alignment distance between <code>phonetic_street_key()</code> of x and y to consider a possible match
name_fuzzy_dist	integer; maximum optimized string alignment distance between @name of x and y to consider a possible match
number_fuzzy_dist	integer; maximum optimized string alignment distance between <code>addr_number</code> strings in x and y to consider a possible match.
match_street_type	character; how to compare street pretype and posttype when selecting street candidates. "exact" requires pretype to match pretype and posttype to match posttype; "compatible" treats blank type fields as unknown but rejects candidates when known type information conflicts; "ignore" does not use street type fields when selecting candidates.
match_street_directional	character; how to compare street predirectional and postdirectional when selecting street candidates. "exact" requires predirectional to match predirectional and postdirectional to match postdirectional; "swap" also permits predirectional to match postdirectional and postdirectional to match predirectional; "ignore" does not use street directional fields when selecting candidates.
progress	logical; show reference-preparation timing and a progress bar while preparing raw y or processing matched ZIP groups?

Details

Prepared indexes record the `addr` canonicalization format. An index serialized by an older package version must be rebuilt with `addr_match_prepare()` before it can be used for matching.

Value

An `addr` vector, the same length as x, containing the selected match in y for each `addr` in x. Partial matches are returned with matched ZIP code and/or street fields filled when later stages do not match.

Examples

```
the_addr <- nad_example_data(match_prepared = TRUE)
my_addr <- as_addr(
  c(
    "2700 Alice St 45222",
    "10623 Springfield Pike 45215",
    "173 Wuhlper Ave 45220",
    "12176 8th Ave 45249",
    "12176 7th Ave 45249",
    "10 W 14th St 45202",
    "10 Oak Rd 45241"
  )
)
```

```

addr_match(my_addr, the_addr)

addr_match(
  my_addr,
  the_addr,
  zip_variants = FALSE,
  name_phonetic_dist = 0L,
  name_fuzzy_dist = 0L,
  number_fuzzy_dist = 0L,
  match_street_type = "ignore",
  match_street_directional = "ignore"
)

my_addr <- as_addr(voter_addresses()[1:25])

d <- addr_match(my_addr, the_addr)
d

addr_match_stage(d)

```

addr_match_stage	<i>Classify addr match stage</i>
------------------	----------------------------------

Description

Classify an addr vector into the staged outcomes returned by `addr_match()`: no match, ZIP-only match, ZIP-plus-street match, or ZIP-plus-street-plus-number match.

Usage

```
addr_match_stage(x, strict = TRUE)
```

Arguments

<code>x</code>	addr vector to classify
<code>strict</code>	logical; require <code>x</code> to follow the partial-result structure produced by <code>addr_match()</code> ? If FALSE, classification is based only on the deepest non-missing core component (<code>@place@zipcode</code> , <code>@street@name</code> , <code>@number@digits</code>).

Value

an ordered factor with levels none, zip, street, number

Examples

```

y <- as_addr(c(
  "10 MAIN ST CINCINNATI OH 45220",
  "11 MAIN ST CINCINNATI OH 45220",
  "10 MAIN ST CINCINNATI OH 45229"
))
x <- as_addr(c(
  "99 MAIN ST CINCINNATI OH 45220",
  "10 OAK ST CINCINNATI OH 45220",
  "10 MAIN ST CINCINNATI OH 45103"
))

out <- addr_match(x, y)
addr_match_stage(out)

```

addr_number

addr classes

Description

The structures for `addr()` and the `addr_` classes are derived as a subset of the United States Thoroughfare, Landmark, and Postal Address Data Standard that is relevant for residential, numbered thoroughfare addresses:

```

Address
+-- AddressNumber
|   +-- AddressNumberPrefix
|   +-- AddressNumber
|   +-- AddressNumberSuffix
+-- StreetName
|   +-- StreetNamePreModifier
|   +-- StreetNamePreDirectional
|   +-- StreetNamePreType
|   +-- StreetName
|   +-- StreetNamePostType
|   +-- StreetNamePostDirectional
+-- Place
    +-- PlaceName
    +-- StateName
    +-- ZipCode

```

`addr()` combines `addr_number()`, `addr_street()`, and `addr_place()` into a single `addr` vector:

```

<addr>
@ number: <addr_number>
.. @ prefix
.. @ digits

```

```

.. @ suffix
@ street: <addr_street>
.. @ predirectional
.. @ premodifier
.. @ pretype
.. @ name
.. @ posttype
.. @ postdirectional
@ place : <addr_place>
.. @ name
.. @ state
.. @ zipcode

```

Usage

```

addr_number(
  prefix = NA_character_,
  digits = NA_character_,
  suffix = NA_character_
)

addr_street(
  predirectional = NA_character_,
  premodifier = NA_character_,
  pretype = NA_character_,
  name = NA_character_,
  posttype = NA_character_,
  postdirectional = NA_character_,
  map_posttype = TRUE,
  map_directional = TRUE,
  map_pretype = TRUE,
  map_ordinal = TRUE
)

addr_place(
  name = NA_character_,
  state = NA_character_,
  zipcode = NA_character_,
  map_state = TRUE
)

addr(number = addr_number(), street = addr_street(), place = addr_place())

```

Arguments

prefix	address number prefix, often a fractional or grid component
digits	primary street number for the address; must be between 0 and 999999
suffix	address number suffix, often a letter or unit-like component

predirectional	direction before the street name
premodifier	descriptive modifier before the street name
pretype	street type or classification before the street name
name	street name, or city/town/municipality name for addr_place()
posttype	street type or classification after the street name
postdirectional	direction after the street name
map_posttype	logical; map posttype to abbreviations?
map_directional	logical; map pre- and post-directional to abbreviations?
map_pretype	logical; map pretype to abbreviations?
map_ordinal	logical; map ordinal street names to abbreviations?
state	state or territory abbreviation
zipcode	ZIP code (must be five digits not starting with "000")
map_state	logical; map state to abbreviations?
number	an addr_number vector
street	an addr_street vector
place	an addr_place vector

Details

All field values must be character vectors of at least length one (including missing values). Length-one fields are recycled to match the length of other fields.

All letters in stored `addr` and `addr_` fields are converted to uppercase. The `map_*` arguments control abbreviation mapping, not this uppercase canonicalization.

Value

An `addr`, `addr_number`, `addr_street`, or `addr_place` vector

Examples

```
# define a new addr_number vector
addr_number(digits = "290")
addr_number(prefix = "N", digits = "290", suffix = "A")

# define a new addr_street vector
addr_street(name = "Burnet", posttype = "Ave")

# street names are automatically mapped to abbreviations
addr_street(predirectional = "North", name = "Fifth", posttype = "Street")

# define a new addr_place vector
addr_place(name = "Cincinnati", state = "OH", zipcode = "45220")

# define a new addr vector
```

```

addr(
  addr_number(digits = "290"),
  addr_street(name = "Burnet", posttype = "Ave"),
  addr_place(name = "Cincinnati", state = "OH", zipcode = "45229")
)

# define a more complicated addr vector
# and explicitly specify empty components to avoid NA
addr(
  addr_number(prefix = "", digits = "200", suffix = ""),
  addr_street(
    predirectional = "west",
    premodifier = "Old",
    pretype = "US",
    name = "50",
    posttype = "avenue",
    postdirectional = "east",
    map_directional = TRUE,
    map_pretype = TRUE,
    map_posttype = TRUE
  ),
  addr_place(name = "Cincinnati", state = "ohio", zipcode = "45220")
)

# addr_* vectors are recycled and omitted fields are missing
addr(
  addr_number(digits = c("290", "200", "3333", "111")),
  addr_street(
    name = c("Burnet", "Main", "Ludlow", "State Route 32"),
    posttype = c("Ave", "St", "Ave", NA_character_)
  ),
  addr_place(name = "Cincinnati", state = "OH")
)

```

as_addr

Coerce to addr

Description

as_addr() converts other objects into addr() vectors. See ?addr for more details on its structure.

Usage

```
as_addr(x, ...)
```

Arguments

x	object to coerce to an addr vector
...	additional arguments passed to methods

Methods implemented for

- **character:** will be cleaned (if `clean = TRUE`) with `clean_address_text()` and then tagged using `usaddress_tag()`; tags are normalized to abbreviations by passing all `map_*` arguments to `addr_street()` or `addr_place()`; ZIP codes parsed with more than five characters are truncated with a warning, and malformed parsed ZIP codes are set to missing with a warning; non-numeric characters in parsed address number digits will be removed with a warning; parsed address number digits greater than 999999 are truncated to the first six digits with a warning
- **data.frame:** must have columns named according to fields in `addr_number()`, `addr_street()`, or `addr_place()`; also passes the `map_*` arguments to `addr_street()` and `addr_place()`
- **addr:** returned as-is when already uppercase; legacy mixed-case objects are reconstructed in the current uppercase canonical form

Legacy `addr` vectors loaded from user-owned serialized files can be upgraded explicitly with `as_addr(readRDS(...))`. The user-owned file is not rewritten automatically.

Examples

```
as_addr(voter_addresses()[1:25])

data.frame(
  number_digits = c("290", "200"),
  street_name = c("Burnet", "Main"),
  street_posttype = c("Ave", "St"),
  place_name = c("Cincinnati", "Cincinnati"),
  place_state = c("OH", "OH"),
  place_zipcode = c("45229", "45220"),
  stringsAsFactors = FALSE
)|>
as_addr()
```

<code>clean_address_text</code>	<i>Clean address text</i>
---------------------------------	---------------------------

Description

Remove excess whitespace and keep only letters, numbers, #, and -.

Usage

```
clean_address_text(x)
```

Arguments

`x` a vector of address character strings

Value

a vector of cleaned addresses

Examples

```
clean_address_text(c(
  "3333 Burnet Ave Cincinnati OH 45219",
  "33_33 Burnet Ave. Cincinnati OH 45219",
  "33\\33 B\"urnet Ave; Ci!ncinn&*ati OH 45219",
  "3333 Burnet Ave Cincinnati OH 45219",
  "33_33 Burnet Ave. Cincinnati OH 45219"
))
```

county_fips_lookup	<i>Translate county names and county FIPS identifiers</i>
--------------------	---

Description

county_fips_lookup() uses a package-internal reference derived from the 2025 U.S. Census county adjacency file to translate between county names, state abbreviations, and 5-digit county FIPS identifiers.

Name lookups accept either the full county-equivalent label (for example, "Orleans Parish") or a shortened form with common suffixes removed (for example, "Orleans"). If a shortened form is ambiguous within a state, the function errors and asks for the full county-equivalent name or the 5-digit FIPS identifier.

Usage

```
county_fips_lookup(county, state = NULL)
```

Arguments

county	character, length one; either a county name or a 5-digit county FIPS identifier
state	character, length one; state abbreviation or full state name; required when county is a name, ignored when county is already a 5-digit FIPS identifier

Value

A tibble with one row and columns county, county_full, state, and county_fips.

Examples

```
county_fips_lookup("Hamilton", "OH")
county_fips_lookup("Hamilton", "Ohio")
county_fips_lookup("39061")
```

elh_data	<i>Example line-one addresses</i>
----------	-----------------------------------

Description

The Cincinnati Eviction Hotspots data was downloaded from [Eviction Labs](#) and contains characteristics of the top 100 buildings that are responsible for about 25% of all eviction filings in Cincinnati (from their "current through 8-31-2024" release).

Usage

```
elh_data()
```

Details

<https://evictionlab.org/eviction-tracking/cincinnati-oh/>

Value

a tibble with 100 rows and 9 columns

Examples

```
elh_data()
```

fuzzy_match	<i>Fuzzy match</i>
-------------	--------------------

Description

Fuzzy match strings in x to strings in y using optimized string alignment (OSA) distance and ignoring capitalization.

Usage

```
fuzzy_match(x, y, osa_max_dist = 1, prefilter = c("none", "psk"))
```

Arguments

- | | |
|--------------|---|
| x | character vector to match |
| y | character vector to match to |
| osa_max_dist | maximum OSA distance to consider a match; Inf is a special case that avoids computing string distance by returning all of y instead of just the best match or matches in y. |
| prefilter | method used to prefilter y before computing OSA distances; "none" does nothing, and "psk" removes values in y that do not share a <code>phonetic_street_key()</code> with any value in x. |

Details

If multiple strings in *y* are tied for the minimum OSA distance from a string in *x*, all of their indices are included in the return value.

Value

a list of integer vectors representing the position of the best matching string(s) in *y* for each string in *x*

Examples

```
my_names <-
  c("Pinye", "Pine", "Oalck", "Sunset", "Riverbend", "Greenfield")
the_names <-
  c("Piney", "Pine", "Oak", "Cheshire", "Greenfield", "Maple", "Elm")
matches <- fuzzy_match(my_names, the_names, osa_max_dist = 1)
matches

lapply(matches, \(i) the_names[i])

x <- as_addr(voter_addresses()[1:25])@street@name
y <- unique(nad_example_data()$nad_addr@street@name)
system.time(fuzzy_match(x, y))
# larger vectors see a speedup when using
# phonetic_street_key as a prefilter
# but may miss potential matches that are within
# osa_max_dist of each other, but did not have
# identical phonetic codes (e.g., "woolper" and "woopler")
system.time(fuzzy_match(x, y, prefilter = "psk"))
```

geocode

Geocode addr vectors with Census TIGER address features

Description

`geocode()` geocodes `addr` vectors using Census TIGER address features (see `?taf`) by:

1. searching for a matching street (see `?match_addr_street`) in the input ZIP code, then in enabled place-derived and typographical ZIP candidates
2. using the address number to select the best address feature range and side of the street (even/odd), breaking ties on smallest width and spread
3. linearly interpolating a geographic point along the best range line based on the actual and potential range of address numbers
4. offsetting the interpolated point from the range line perpendicularly

Only matched input addresses return non-missing matched ZIP code and street values. Missing or unmatched ZIP codes return missing matched ZIP code, street, geography, and s2 cell values. If no matching street has a valid range for the address number, the best street-only match is retained and only the geography and s2 cell values return NA.

Usage

```

geocode(
  x,
  name_phonetic_dist = 1L,
  name_fuzzy_dist = 2L,
  match_street_type = c("exact", "compatible", "ignore"),
  match_street_directional = c("exact", "swap", "ignore"),
  place_zip_variants = TRUE,
  place_zip_variant = c("place", "county-sub"),
  zip_variants = TRUE,
  zip_variant = c("minus1", "plus1", "sub5", "sub4", "swap"),
  year = as.character(2025:2011),
  version = "v2",
  taf_install = TRUE,
  taf_redownload = FALSE,
  offset = 10L,
  add_s2_cell = TRUE,
  progress = interactive()
)

geocode_zip(
  x,
  offset = 10L,
  name_phonetic_dist = 1L,
  name_fuzzy_dist = 2L,
  match_street_type = c("exact", "compatible", "ignore"),
  match_street_directional = c("exact", "swap", "ignore"),
  zip_variants = TRUE,
  zip_variant = c("minus1", "plus1", "sub5", "sub4", "swap"),
  year = as.character(2025:2011),
  version = "v2",
  taf_install = TRUE,
  taf_redownload = FALSE,
  progress_callback = NULL,
  taf_check = TRUE,
  place_zip_variants = TRUE,
  place_zip_variant = c("place", "county-sub")
)

```

Arguments

x an addr vector (?as_addr)

name_phonetic_dist integer; maximum optimized string alignment distance between `phonetic_street_key()` of x and y to consider a possible match

name_fuzzy_dist integer; maximum optimized string alignment distance between @name of x and y to consider a possible match

<code>match_street_type</code>	character; how to compare street pretype and posttype when selecting street candidates. "exact" requires pretype to match pretype and posttype to match posttype; "compatible" treats blank type fields as unknown but rejects candidates when known type information conflicts; "ignore" does not use street type fields when selecting candidates.
<code>match_street_directional</code>	character; how to compare street predirectional and postdirectional when selecting street candidates. "exact" requires predirectional to match predirectional and postdirectional to match postdirectional; "swap" also permits predirectional to match postdirectional and postdirectional to match predirectional; "ignore" does not use street directional fields when selecting candidates.
<code>place_zip_variants</code>	logical; consider ZCTAs associated with the exact normalized place and state after the input ZIP and before typographical ZIP variants? Defaults to TRUE.
<code>place_zip_variant</code>	nonempty character vector containing "place" and/or "county-sub". "place" uses Census places and "county-sub" uses Census county subdivisions. Requested order determines precedence; the default uses both in that order.
<code>zip_variants</code>	logical; consider typographical variants of the input ZIP code after exact and enabled place-derived candidates? Defaults to TRUE.
<code>zip_variant</code>	nonempty character vector selecting typographical ZIP variants; see zipcode_variant() . Requested order determines precedence.
<code>year</code>	character, length 1; vintage of TIGER addrfeat (address feature) files
<code>version</code>	character, length 1; major version of the package and taf dataset schema
<code>taf_install</code>	logical; install missing county TAF files needed for input ZIP codes and all selected place-derived and typographical ZIP candidates before geocoding? If FALSE, geocoding proceeds with installed files only and warns when needed county files are missing.
<code>taf_redownload</code>	logical; replace existing durable managed local copies of TIGER ZIP files when installing missing TAF counties?
<code>offset</code>	number of meters to offset geocode from street line
<code>add_s2_cell</code>	logical; add an <code>s2_cell</code> column computed from matched geographies? Defaults to TRUE; set to FALSE to skip this final computation.
<code>progress</code>	logical; show progress messages and a ZIP-code progress bar while geocoding?
<code>progress_callback</code>	optional callback used internally by <code>geocode()</code> to update progress after ZIP-code reference data is loaded
<code>taf_check</code>	logical; check for missing TAF counties? Used internally by <code>geocode()</code> after checking once for the full input vector.

Details

`geocode_zip()` is the workhorse function and operates on `addr` vectors with the same ZIP code; use `geocode()` to geocode an `addr` vector with multiple ZIP codes by grouping them by ZIP code

and processing serially by default. At a lower level, grouping `addr` vectors by ZIP code and applying `geocode_zip()` facilitates more control (e.g., parallel processing). Before ZIP grouping, `geocode()` deduplicates formatted `addr` values internally and restores the output to the original input order and length. Exact duplicates therefore do not trigger repeated TAF reads, street matching, or range interpolation, so callers usually do not need to call `unique()` themselves for geocoding performance.

ZIP matching uses ordered candidate tiers. The exact input ZIP is first. When `place_zip_variants = TRUE`, ZCTAs whose normalized Census place or county-subdivision name exactly equals the input place and whose state equals the input state are next, in `place_zip_variant` order. Typographical ZIP candidates from `zipcode_variant()` follow in `zip_variant` order when `zip_variants = TRUE`. Place matching is deliberately conservative: it does not use fuzzy matching or infer a ZIP when the input ZIP is missing.

Valid address ranges are preferred over street-only matches across all enabled tiers. Among valid ranges, exact ZIP wins, followed by the requested place geography order and then the requested typographical variant order. Consequently, a valid place-derived or typographical range can beat an earlier street-only match, but an exact-ZIP valid range always wins. If no candidate has a valid range, the earliest tier's best street-only match is retained.

The "place" selector accepts ZCTAs supported by the Census place file or by both relationship files. The "county-sub" selector accepts ZCTAs supported by the county-subdivision file or by both files. A ZCTA supported by both is assigned to the first requested selector so it is tried only once. Input place names are uppercased, whitespace is normalized, and a trailing Census CITY, VILLAGE, TOWN, TOWNSHIP, BOROUGH, or CDP suffix is removed before exact matching. Internal punctuation, diacritics, and other descriptors are preserved.

If the `mirai` package is installed and `mirai` daemons have already been configured by the caller, `geocode()` uses them for ZIP-code-level parallel processing. Otherwise it falls back to sequential processing.

`geocode()` and `geocode_zip()` both download and install tiger address features by county (`?taf_install`) as needed based on the input `addr` ZIP codes and all enabled candidate tiers. TAF install checks run before reading TAF ZIP files so parallel geocoding workers do not try to download county files at the same time. Because both place-derived and typographical ZIP variants are enabled by default, a `geocode` call can consider many ZCTAs and download many county TAF files. Common place/state names can increase this substantially. Disable `place_zip_variants`, `zip_variants`, or both when that broader search and its downloads are not wanted.

Value

A tibble with columns `addr` (the input `addr` vector), `matched_zipcode` (character vector), `matched_street` (`addr_street` vector), and `matched_geography` (`s2_geography` point vector). When `add_s2_cell = TRUE`, the tibble also includes `s2_cell` (`s2_cell` vector).

Examples

```
x <- as_addr(voter_addresses()[1:25])
taf_needed_counties(x)

## Not run:
# for example purposes, only install one county
Sys.setenv("R_USER_DATA_DIR" = tempfile())
```

```

    taf_install("39061", "2025")
    # and geocode without installing other counties
    gcd <- geocode(x, taf_install = FALSE)

    # this is only for example purposes and usually not required; e.g.
    gcd <- geocode(x)

    gcd

    table(geocode_stage(gcd))

    geocode_table(gcd)

    leaflet::leaflet(wk::wk_coords(gcd$matched_geography)) |>
      leaflet::addTiles() |>
      leaflet::addCircleMarkers(lng = ~x, lat = ~y, label = ~feature_id)

## End(Not run)

## Not run:
# use mirai for parallel processing
mirai::daemons(2)
geocode(x)
mirai::daemons(0)

## End(Not run)

```

geocode_stage

Classify geocode stage

Description

Classify geocode results into staged outcomes returned by `geocode()`: no match, street match, or interpolated range match, distinguishing exact ZIP-code matches from ZIP-code variant matches.

Usage

```
geocode_stage(x)
```

Arguments

`x` a data frame returned by `geocode()`

Value

an ordered factor with levels none, street_variant, street, range_variant, range

geocode_table	<i>Convert geocode objects to JSON-safe tables</i>
---------------	--

Description

geocode_table() converts the rich output from [geocode\(\)](#) to a flat table with only JSON-safe column types.

Usage

```
geocode_table(x)
```

Arguments

x a data frame returned by [geocode\(\)](#)

Value

A tibble with atomic columns suitable for JSON serialization. geocode_table() includes the input address, geocode stage, matched ZIP code, matched street, and, when present, S2 cell as character columns.

match_addr_number	<i>Match addr_number vectors</i>
-------------------	----------------------------------

Description

A single addr_number in y is chosen for each addr_number in x. If exact matches (using `as.character()`) are not found, possible matches within `number_fuzzy_dist` are searched for in y. If multiple matches are present in y, the selected match has the lowest absolute numeric difference from `@digits` in x; ties are broken by optimized string alignment (OSA) distance and then by lexicographic order with digits preceding alphabetic characters.

addr_number objects with missing `@digits`, or with empty strings for all of `@prefix`, `@digits`, and `@suffix`, are not matched and returned as missing instead.

Usage

```
match_addr_number(x, y, number_fuzzy_dist = 1L)
```

Arguments

x, y addr_number vectors to match
 number_fuzzy_dist integer; maximum optimized string alignment distance between addr_number strings in x and y to consider a possible match.

Value

An `addr_number` vector, the same length as `x`, containing the selected match in `y` for each element of `x`. Unmatched elements are returned as missing `addr_number()` values.

Examples

```
x <- addr_number(
  prefix = "",
  digits = as.character(c(1, 10, 228, 11, 22, 22, 22, 10, 99897, NA)),
  suffix = ""
)

y <- addr_number(
  prefix = "",
  digits = as.character(c(12, 11, 10, 22)),
  suffix = ""
)

match_addr_number(x, y)

match_addr_number(x, y, number_fuzzy_dist = 0L)
```

match_addr_street	<i>Match addr_street vectors</i>
-------------------	----------------------------------

Description

A single `addr_street` in `y` is chosen for each `addr_street` in `x`. If exact matches (using `as.character()`) are not found, candidate matches are chosen by fuzzy matching on street name (using phonetic street key and street name) and matching the street type and directional components according to `match_street_type` and `match_street_directional`. Ordinal street names use restricted phonetic candidates: an ordinal phonetic key like `#0007` may fuzzy match only to plausible ordinal neighbors such as digit shifts (`#0070`, `#0700`, `#7000`) or same-width substitutions (`#0008`, `#0009`), not arbitrary OSA-distance-one ordinal keys such as `#0017` or `#0077`. If multiple candidates remain after fuzzy matching, the first candidate in `y` is returned.

`addr_street` objects with missing or empty `@name` are not matched and returned as missing instead.

Usage

```
match_addr_street(
  x,
  y,
  name_phonetic_dist = 1L,
  name_fuzzy_dist = 2L,
  match_street_type = c("exact", "compatible", "ignore"),
  match_street_directional = c("exact", "swap", "ignore")
)
```

Arguments

`x, y` `addr_street` vectors to match

`name_phonetic_dist` integer; maximum optimized string alignment distance between `phonetic_street_key()` of `x` and `y` to consider a possible match

`name_fuzzy_dist` integer; maximum optimized string alignment distance between `@name` of `x` and `y` to consider a possible match

`match_street_type` character; how to compare street pretype and posttype when selecting street candidates. "exact" requires pretype to match pretype and posttype to match posttype; "compatible" treats blank type fields as unknown but rejects candidates when known type information conflicts; "ignore" does not use street type fields when selecting candidates.

`match_street_directional` character; how to compare street predirectional and postdirectional when selecting street candidates. "exact" requires predirectional to match predirectional and postdirectional to match postdirectional; "swap" also permits predirectional to match postdirectional and postdirectional to match predirectional; "ignore" does not use street directional fields when selecting candidates.

Value

An `addr_street` vector, the same length as `x`, containing the selected match in `y` for each element of `x`. Unmatched elements are returned as missing `addr_street()` values.

Examples

```
my_streets <- addr_street(
  predirectional = "",
  premodifier = "",
  pretype = "",
  name = c("Beechview", "Vivian", "Springfield", "Round Bottom", "Pfeiffer", "Beachview",
           "Vevan", "Srpingfield", "Square Top", "Pfeffer", "Wuhlper", ""),
  posttype = c("Cir", "Pl", "Pike", "Rd", "Rd", "Cir", "Pl", "Pike", "Rd", "Rd", "Ave", ""),
  postdirectional = ""
)
the_streets <- nad_example_data()$nad_addr@street
match_addr_street(my_streets, the_streets)

toggle_y <- addr_street(
  predirectional = c("E", "", "", "E"),
  premodifier = "",
  pretype = c("", "", "US Hwy", "US Hwy"),
  name = c("14th", "Oak", "Main", "Main"),
  posttype = c("St", "Rd", "Rd", "Rd"),
  postdirectional = c("", "", "", "E"),
  map_pretype = FALSE,
  map_posttype = FALSE,
```

```

    map_directional = FALSE,
    map_ordinal = FALSE
)

# directionals are required by default, so blank "14th St" stays unmatched
format(match_addr_street(
  addr_street(
    predirectional = "",
    premodifier = "",
    pretype = "",
    name = "14th",
    posttype = "St",
    postdirectional = "",
    map_pretype = FALSE,
    map_posttype = FALSE,
    map_directional = FALSE,
    map_ordinal = FALSE
  ),
  toggle_y
))
format(match_addr_street(
  addr_street(
    predirectional = "",
    premodifier = "",
    pretype = "",
    name = "14th",
    posttype = "St",
    postdirectional = "",
    map_pretype = FALSE,
    map_posttype = FALSE,
    map_directional = FALSE,
    map_ordinal = FALSE
  ),
  toggle_y,
  match_street_directional = "ignore"
))

# type can also be ignored during fuzzy street-name matching
format(match_addr_street(
  addr_street(
    predirectional = "",
    premodifier = "",
    pretype = "",
    name = "Oka",
    posttype = "Ave",
    postdirectional = "",
    map_pretype = FALSE,
    map_posttype = FALSE,
    map_directional = FALSE,
    map_ordinal = FALSE
  ),
  toggle_y,
  name_fuzzy_dist = 1L

```

```

))
format(match_addr_street(
  addr_street(
    predirectional = "",
    premodifier = "",
    pretype = "",
    name = "Oka",
    posttype = "Ave",
    postdirectional = "",
    map_pretype = FALSE,
    map_posttype = FALSE,
    map_directional = FALSE,
    map_ordinal = FALSE
  ),
  toggle_y,
  name_fuzzy_dist = 1L,
  match_street_type = "ignore"
))

# compatible type matching allows blanks to stand in for unknown type fields
type_y <- addr_street(
  predirectional = "",
  premodifier = "",
  pretype = c("Ave", "Rd"),
  name = "Main",
  posttype = "",
  postdirectional = "",
  map_pretype = FALSE,
  map_posttype = FALSE,
  map_directional = FALSE,
  map_ordinal = FALSE
)
format(match_addr_street(
  addr_street(
    predirectional = "",
    premodifier = "",
    pretype = "",
    name = "Main",
    posttype = "Ave",
    postdirectional = "",
    map_pretype = FALSE,
    map_posttype = FALSE,
    map_directional = FALSE,
    map_ordinal = FALSE
  ),
  type_y,
  match_street_type = "compatible"
))

# type and directional matching can be relaxed independently
format(match_addr_street(
  addr_street(
    predirectional = "E",

```

```

    premodifier = "",
    pretype = "US Hwy",
    name = "Mian",
    posttype = "Rd",
    postdirectional = "E",
    map_pretype = FALSE,
    map_posttype = FALSE,
    map_directional = FALSE,
    map_ordinal = FALSE
  ),
  toggle_y,
  match_street_type = "ignore",
  name_fuzzy_dist = 1L
))
format(match_addr_street(
  addr_street(
    predirectional = "E",
    premodifier = "",
    pretype = "US Hwy",
    name = "Mian",
    posttype = "Rd",
    postdirectional = "E",
    map_pretype = FALSE,
    map_posttype = FALSE,
    map_directional = FALSE,
    map_ordinal = FALSE
  ),
  toggle_y,
  name_fuzzy_dist = 1L
))
format(match_addr_street(
  addr_street(
    predirectional = "E",
    premodifier = "",
    pretype = "US Hwy",
    name = "Mian",
    posttype = "Rd",
    postdirectional = "E",
    map_pretype = FALSE,
    map_posttype = FALSE,
    map_directional = FALSE,
    map_ordinal = FALSE
  ),
  toggle_y,
  name_fuzzy_dist = 1L,
  match_street_type = "exact",
  match_street_directional = "exact"
))

```

Description

A single ZIP code in *y* is chosen for each ZIP code in *x*. By default, if exact matches are not found, common variants of ZIP codes in *x* are searched for in *y* (`?zipcode_variant`). If multiple variants are present in *y*, the selected match has the lowest absolute numeric difference from the ZIP code in *x*; ties are broken by OSA string distance and then by the minimum number.

Usage

```
match_zipcodes(
  x,
  y,
  zip_variants = TRUE,
  zip_variant = c("minus1", "plus1", "sub5", "sub4", "swap")
)
```

Arguments

<code>x, y</code>	character vectors of ZIP codes to match
<code>zip_variants</code>	logical; fuzzy match to common variants of <i>x</i> in <i>y</i> ?
<code>zip_variant</code>	character vector; zipcode variant types to use when <code>zip_variants</code> is TRUE; see <code>?zipcode_variant</code>

Value

A character vector, the same length as *x*, containing the selected match in *y* for each ZIP code in *x*.

Examples

```
match_zipcodes(
  c("45222", "45219", "45219", "45220", "45220", "", NA),
  c("42522", "45200", "45219", "45221", "45223", "45321", "")
)

match_zipcodes(
  c("45222", "45219", "45219", "45220", "45220", "", NA),
  c("42522", "45200", "45219", "45221", "45223", "45321", ""),
  zip_variants = FALSE
)
```

Description

The U.S. Department of Transportation partners with address programs from state, local, and tribal governments to compile their authoritative data into a database. Find more information in the source data portal: <https://data.transportation.gov/d/fc2s-wawr>

`nad()` reads processed NAD data for one county, installing that county from source data managed by `nad_download()` when needed. Counties can be identified either by county name plus state, or by a 5-digit county FIPS identifier. County names and state abbreviations are resolved internally and determine the processed-data path and source filter. The revision 23 source is a roughly 7.6 GB compressed archive containing a roughly 41 GB comma-delimited text member. `nad_catalog()` reads the packaged revision 23 catalog of source-available counties and is consulted before extracting a missing county.

Processed county Parquet files are created one county at a time with `nad_install()` or on first use with `nad()`. The compressed national source is managed exclusively by `stow()` beneath `stow::stow_path(package = "addr", subdir = "nad")`. Derived county Parquet files are a separate Hive-partitioned dataset beneath `file.path(tools::R_user_dir("addr", "data"), "v2", "nad", "23")`, organized by state and `county_fips`. `nad_manifest()` inventories those county files from `v2/nad_manifest/23/counties.parquet`. Set `R_USER_DATA_DIR` to relocate both areas while retaining their source-versus-processed-data separation.

Usage

```
nad(
  county,
  state = NULL,
  version = 23L,
  refresh_binary = c("yes", "no", "force"),
  refresh_source = c("no", "yes", "force")
)

nad_install(
  county,
  state = NULL,
  version = 23L,
  overwrite = FALSE,
  refresh_source = c("no", "yes", "force")
)

nad_download(version = 23L, refresh_source = c("yes", "no", "force"))
```

Arguments

<code>county</code>	character, length one; county name or 5-digit county FIPS identifier
<code>state</code>	character, length one; name or abbreviation of state. Required when <code>county</code> is a county name; ignored when <code>county</code> is a 5-digit county FIPS identifier
<code>version</code>	integer, length one; NAD revision to use. Only revision 23 is supported.

refresh_binary	character, length one; choose how to refresh a processed NAD county Parquet file if not already present; "yes" creates a missing file, "no" requires it to exist, and "force" recreates it
refresh_source	character, length one; choose how to refresh the compressed national source; "no" requires the stow-managed source to exist, "yes" downloads it if missing, and "force" redownloads it
overwrite	logical, length one; overwrite an existing processed county Parquet file?

Details

The revision 23 comma-delimited flat source archive is downloaded from the transportation.gov data portal: <https://data.transportation.gov/d/fc2s-wawr>. `nad_download(version = 23L)` installs the pinned compressed archive as a durable managed local copy using `stow::stow()`. County installation streams the nationwide text member directly from that archive, retains the requested county, and writes a separate processed Parquet file. The roughly 41 GB text member is never unpacked on disk. Before downloading, review the source metadata and disclaimer in the data portal.

Investigate individual address points in the online viewer: <https://usdot.maps.arcgis.com/apps/instant/portfolio/index.html?appid=59f7e4fb71994d13b61f424e21a6cffe>

The NAD does not distinguish between empty and missing address components. When reading into R, all missing address components are replaced with an empty string ("") *except* for address number (digits), street name, and ZIP code. Addresses with malformed ZIP codes are removed.

Value

`nad()` returns a tibble for one county. `nad_install()` invisibly returns the installed county FIPS identifier.

Examples

```
# explicitly download source data, then create county output on first read
## Not run:
# manually install the compressed revision 23 flat source archive
nad_download()
# manually build a county Parquet file
nad_install("39017")

# read files by name or fips id into R,
# nad will be downloaded and counties installed as needed
nad("Butler", "OH")
nad("39017")

## End(Not run)

# small packaged revision 23 fixture derived from Hamilton County, OH
nad_example_data()
```

nad_catalog

Read the National Address Database county catalog

Description

nad_catalog() reads the revision-specific catalog installed with addr. The catalog describes counties that can be mapped unambiguously from the pinned NAD source to Census county FIPS identifiers. It is distinct from nad_manifest(), which inventories county Parquet files installed in the current user's data directory.

Usage

```
nad_catalog(version = 23L)
```

Arguments

version integer, length one; NAD revision to use. Only revision 23 is supported.

Details

NAD23 does not include a county FIPS field. For six county/independent-city pairs, the source uses an unsuffixed label for the county and a separate explicit city label for the independent city. The catalog preserves those exact source labels while mapping each one to its distinct county FIPS code. source_row_count counts raw NAD records before county installation removes records with malformed ZIP codes.

Value

A tibble with one row per source-available county and columns county_fips, state, county, source_county, source_row_count, and nad_revision.

Examples

```
nad_catalog()
```

nad_dataset

Open installed National Address Database counties as an Arrow dataset

Description

nad_dataset() opens every installed county Parquet file as one lazy Arrow dataset. The storage schema contains primitive columns suitable for filtering, projection, and collection with dplyr. state and county_fips are Hive partition columns. Use nad() when reconstructed addr and s2 columns are needed for one county.

Usage

```
nad_dataset(version = 23L)
```

Arguments

version integer, length one; NAD revision to use. Only revision 23 is supported.

Value

An Arrow FileSystemDataset.

Examples

```
## Not run:
nad_dataset() |>
  dplyr::filter(state == "OH", county_fips == "39017") |>
  dplyr::select(uuid, address_number, street_name, longitude, latitude) |>
  dplyr::collect()

## End(Not run)
```

nad_example_data	<i>Example National Address Database addresses</i>
------------------	--

Description

A small baked example fixture containing Hamilton County, Ohio rows from revision 23 of the National Address Database. See `?nad` for more information about the National Address Database.

The fixture was derived once from the Hamilton County revision 23 source and is not intended to be regenerated by package users. Use `nad()` when you need complete county data.

Usage

```
nad_example_data(match_prepared = FALSE)
```

Arguments

match_prepared logical; return the example data preprocessed with `addr_match_prepare()`?

Value

If `match_prepared = FALSE`, a tibble with 5,000 rows and 7 columns. If `match_prepared = TRUE`, an `addr_match_index`.

Examples

```
nad_example_data()
nad_example_data(match_prepared = TRUE)
```

nad_manifest	<i>Inventory installed National Address Database counties</i>
--------------	---

Description

`nad_manifest()` reads the local manifest written as county Parquet files are installed. The Parquet path remains the primary existence check used by `nad()`; the manifest provides a compact inventory and integrity metadata for validation.

Usage

```
nad_manifest(version = 23L, validate = FALSE)
```

Arguments

<code>version</code>	integer, length one; NAD revision to inventory. Only revision 23 is supported.
<code>validate</code>	logical, length one; validate every manifest row against its county Parquet file, including its schema, row count, byte size, and SHA-256 digest? This can be slow for a large inventory.

Value

A tibble with one row per installed county and columns `county_fips`, `state`, `county`, `row_count`, `size_bytes`, `sha256`, `nad_revision`, and `installed_at`.

Examples

```
nad_manifest()
```

phonetic_street_key	<i>Convert street names into phonetic matching keys</i>
---------------------	---

Description

Ordinal street names (e.g., "11TH", "5TH") are encoded as zero-padded numeric identifiers with a special prefix, while non-ordinal street names are encoded using a Soundex phonetic code (see `?stringdist::phonetic`). Ordinal words (e.g., "Eleventh", "Fifth") are detected and converted automatically. Each phonetic key is exactly four characters long.

Usage

```
phonetic_street_key(x)
```

Arguments

<code>x</code>	character vector
----------------	------------------

Value

character vector

Examples

```
phonetic_street_key(
  c("MEADOWLARK", "TOWNSEND", "IMMACULATE", "7TH", "WERK",
    "PAXTON", "5th", "BURNET", "FIFTH", "CLIFTON")
)
```

taf

Read TIGER Address Features into R

Description

`taf()` reads installed TIGER Address Features for one or more ZIP codes. It reconstructs the `county_fips`, `s2_geography`, and `addr_street` vectors in the returned data frame. `taf_install()` installs one county's processed Parquet files from the stow-managed TIGER sources. Use `taf_dataset()` to query all installed flat files lazily with Arrow.

Usage

```
taf_install(
  county,
  year = as.character(2025:2011),
  version = "v2",
  overwrite = FALSE,
  redownload = FALSE
)
```

```
taf(x, map = TRUE, year = as.character(2025:2011), version = "v2")
```

Arguments

<code>county</code>	character, length 1; county FIPS code
<code>year</code>	character, length 1; vintage of TIGER <code>addrfeat</code> (address feature) files
<code>version</code>	character, length 1; major version of the package and taf dataset schema
<code>overwrite</code>	logical, length 1; overwrite an existing county install?
<code>redownload</code>	logical, length 1; replace existing durable managed local copies of TIGER ZIP files?
<code>x</code>	character vector of five-digit ZIP codes
<code>map</code>	logical, length 1; map street tags read from the TAF data (type, directional, ordinal) when converting to <code>addr_street()</code> vector?

Details

About 6% of ADDRFEAT rows do not have a county-local primary FEATNAMES match by LINEARID. In these cases, street tags are parsed from the ADDRFEAT full name, and the street_tag_parsed column is set to TRUE.

Value

taf() returns a tibble with LINEARID, FULLNAME, side, ZIP, FROMHN, TOHN, PARITY, OFFSET, s2_geography, addr_street, county_fips, and street_tag_parsed columns. taf_install() invisibly returns the installed county FIPS identifier.

Examples

```
## Not run:
  Sys.setenv("R_USER_DATA_DIR" = tempfile())
  taf_install("39061", "2025")

## End(Not run)
## Not run:
  Sys.setenv("R_USER_DATA_DIR" = tempfile())
  taf_install("39061", "2025")
  taf(c("45249", "45230", "45220"))

## End(Not run)
```

taf_catalog	<i>Read TIGER address feature ZIP/county catalog</i>
-------------	--

Description

taf_catalog() reads a TIGER-derived catalog of ZIP codes present in each county’s TIGER address feature file for a specific year and addr TAF schema version. The catalog is installed with the package and is used to plan which county TAF files may be needed for a set of ZIP codes. It is separate from the local install manifest, which records only files installed on the current machine.

Usage

```
taf_catalog(year = as.character(2025:2011), version = "v2")
```

Arguments

- year character, length 1; vintage of TIGER addrfeat (address feature) files
- version character, length 1; major version of the package and taf dataset schema

Value

a tibble with county_fips, ZIP, zip3, zip2, and n_ranges columns

Examples

```
taf_catalog("2025")
```

taf_dataset

Open installed TIGER Address Features as an Arrow dataset

Description

taf_dataset() uses the arrow package to open the Hive-partitioned Parquet dataset of TIGER address features in the addr user data directory. Arrow FileSystemDataset objects are database-like backends for larger-than-memory datasets and support dplyr syntax for data manipulation; see https://arrow.apache.org/docs/r/articles/data_wrangling.html. Other TAF helpers such as taf_catalog(), taf_install(), and taf() use nanoparquet directly for flat parquet file reads and writes. Arrow is only required for the advanced dataset interface returned by taf_dataset().

Usage

```
taf_dataset(year = as.character(2025:2011), version = "v2")
```

Arguments

year	character, length one; vintage of TIGER addrfeat (address feature) files
version	character, length one; major version of the package and taf dataset schema

Value

a Dataset R6 object (see ?arrow::open_dataset); use dplyr verbs to query the data and get results, see examples

Examples

```
## Not run:
Sys.setenv("R_USER_DATA_DIR" = tempfile())
taf_install("39061", "2025")

if (requireNamespace("arrow", quietly = TRUE) &&
    requireNamespace("dplyr", quietly = TRUE)) {
  taf_dataset()

  # find top ten most frequent street name-posttype combinations
  taf_dataset() |>
    dplyr::group_by(street_name, street_posttype) |>
    dplyr::summarize(
      n_zips = dplyr::n_distinct(ZIP),
      n_ranges = dplyr::n(),
      .groups = "drop"
    ) |>
    dplyr::arrange(dplyr::desc(n_zips), dplyr::desc(n_ranges)) |>
```

```

      dplyr::collect() |>
      dplyr::slice(1:10)
    }

## End(Not run)

```

taf_manifest

Inventory installed TIGER Address Feature files

Description

taf_manifest() reads the local inventory written as county TAF files are installed. Each row represents one installed county-ZIP Parquet file. Set validate = TRUE to verify the manifest schema and every inventoried file, including its storage schema, row count, byte size, and SHA-256 digest.

Usage

```
taf_manifest(year = as.character(2025:2011), version = "v2", validate = FALSE)
```

Arguments

year	character, length 1; vintage of TIGER addrfeat (address feature) files
version	character, length 1; major version of the package and taf dataset schema
validate	logical, length one; validate every manifest row and reject missing, untracked, unreadable, or modified Parquet files?

Details

The local manifest is distinct from [taf_catalog\(\)](#). The catalog describes every county-ZIP combination available from TIGER, while the manifest records only files installed on the current machine.

Value

A tibble with one row per installed county-ZIP file and columns county_fips, ZIP, zip3, zip2, n_ranges, size_bytes, sha256, taf_year, taf_version, and installed_at.

Examples

```

## Not run:
taf_manifest()
taf_manifest(validate = TRUE)

## End(Not run)

```

taf_needed_counties	<i>Find and install TAF counties needed for ZIP codes</i>
---------------------	---

Description

taf_needed_counties() uses taf_catalog() to identify county TAF files that may contain address ranges for ZIP codes in x. For addr input it includes selected place, county-subdivision, and typographical ZIP candidates when requested. Character input contains ZIP codes only, so place-derived candidates cannot be inferred. taf_ensure() installs any needed counties that are not already present in the local TAF manifest.

Usage

```
taf_needed_counties(
  x,
  year = as.character(2025:2011),
  version = "v2",
  zip_variants = TRUE,
  zip_variant = c("minus1", "plus1", "sub5", "sub4", "swap"),
  place_zip_variants = TRUE,
  place_zip_variant = c("place", "county-sub")
)

taf_ensure(
  x,
  year = as.character(2025:2011),
  version = "v2",
  zip_variants = TRUE,
  zip_variant = c("minus1", "plus1", "sub5", "sub4", "swap"),
  place_zip_variants = TRUE,
  place_zip_variant = c("place", "county-sub"),
  redownload = FALSE
)
```

Arguments

x	an addr vector (?as_addr) or character vector of ZIP codes
year	character, length 1; vintage of TIGER addrfeat (address feature) files
version	character, length 1; major version of the package and taf dataset schema
zip_variants	logical; fuzzy match to common variants of x in y?
zip_variant	character vector; zipcode variant types to use when zip_variants is TRUE; see ?zipcode_variant
place_zip_variants	logical; for addr input, consider ZCTAs associated with the normalized input place or county subdivision?

place_zip_variant	nonempty character vector containing "place" and/or "county-sub"; requested order determines precedence
redownload	logical, length 1; replace existing durable managed local copies of TIGER ZIP files?

Value

taf_needed_counties() returns a tibble with catalog columns plus source_zip and source_zip_variant.
taf_ensure() invisibly returns the subset of needed counties that were missing before installation.

Examples

```
taf_needed_counties(as_addr("10 MAIN ST CINCINNATI OH 45220"))
```

tag_usaddress	Tag US addresses
---------------	------------------

Description

Addresses are tagged using the usaddress conditional random field in a rust port of usaddress. Possible address labels include:

- AddressNumberPrefix
- AddressNumberSuffix
- AddressNumber
- BuildingName
- CornerOf
- IntersectionSeparator
- LandmarkName
- NotAddress
- OccupancyIdentifier
- OccupancyType
- PlaceName
- Recipient
- StateName
- StreetNamePostDirectional
- StreetNamePostType
- StreetNamePreDirectional
- StreetNamePreModifier
- StreetNamePreType
- StreetName

- SubaddressIdentifier
- SubaddressType
- USPSBoxGroupID
- USPSBoxGroupType
- USPSBoxID
- USPSBoxType
- ZipCode

Find more information about the definitions at <https://www.fgdc.gov/standards/projects/address-data>

Usage

```
tag_usaddress(x = NA_character_, clean = TRUE)
```

Arguments

x	character string of addresses
clean	logical; clean address text with <code>clean_address_text()</code> before tagging?

Value

a list of vectors of named address tags

Examples

```
tag_usaddress(
  c("290 Ludlow Avenue Apt 2 Cincinnati OH 45220",
    "3333 Burnet Ave Cincinnati Ohio 45219",
    "120 North Main Street, Greenville, SC 29601",
    "200 Southwest North Street, Topeka, KS 66603",
    "215 Highway 88 Road, Jackson, CA 95642"
  )
)

# edge cases!
tag_usaddress(
  c(
    "1600 Pennsylvania Avenue NW, Washington, DC 20500", # post-directional quadrant
    "1 Infinite Loop, Cupertino, CA 95014", # corporate campus street name
    "210 East 400 South, Salt Lake City, UT 84111", # grid addressing (Utah)
    "N6W23001 Bluemound Road, Wauwatosa, WI 53226", # address number prefix grid (Wisconsin)
    "350 Fifth Avenue, New York, NY 10118", # ordinal street name
    "4059 Mt Lee Drive, Hollywood, CA 90068", # abbreviated street element
    "233 South Wacker Drive, Chicago, IL 60606", # pre-directional
    "700 Exposition Park Drive, Los Angeles, CA 90037", # multi-word street name
    "2 South Biscayne Boulevard, Miami, FL 33131" # directional + boulevard
  )
)
```

tiger_addr_feat	<i>Get s2_geography for tiger street ranges</i>
-----------------	---

Description

TIGER address features (street address ranges) are read from compressed addrfeat (address feature) shapefiles for each county and Census vintage. If not already present, compressed addrfeat shapefiles are downloaded from the Census TIGER HTTPS endpoint by default as durable managed local copies in addr's stow/tiger_addr_feat directory. Files from the former unmanaged TIGER layout are not searched.

When reading into R, the data is converted to one row per street side (L/R) for use by taf_install().

Usage

```
tiger_addr_feat(county, year, redownload = FALSE)
```

Arguments

county	character string of county FIPS identifier
year	character year of the Census TIGER/Line product
redownload	logical, length 1; replace the durable managed local copy of the TIGER ZIP file?

Value

a tibble with LINEARID, FULLNAME, side, ZIP, FROMHN, TOHN, PARITY, OFFSET, and s2_geography columns

Examples

```
## Not run:
tiger_addr_feat("39061", "2025")

## End(Not run)
```

tiger_feat_names	<i>Get names for tiger street ranges</i>
------------------	--

Description

TIGER primary feature names are read from compressed feature-name databases for each county and Census vintage. If not already present, compressed addrfeat (address feature) shapefiles are downloaded from the Census TIGER HTTPS endpoint by default as durable managed local copies in addr's stow/tiger_feat_names directory. Files from the former unmanaged TIGER layout are not searched.

When reading into R, the data is filtered to addressable MTFCCs (S1100, S1200, S1400, S1640) that have a name.

Usage

```
tiger_feat_names(county, year, redownload = FALSE)
```

Arguments

- county character string of county FIPS identifier
- year character year of the Census TIGER/Line product
- redownload logical, length 1; replace the durable managed local copy of the TIGER ZIP file?

Value

a tibble with unique LINEARID and addr columns

Examples

```
## Not run:
tiger_feat_names("39061", "2025")

## End(Not run)
```

voter_addresses	<i>Example addresses</i>
-----------------	--------------------------

Description

voter_addresses() returns a small baked example character vector derived from real-world addresses originally downloaded from the Hamilton County, Ohio voter registration database on 2024-09-12. The original export URL was <https://votehamiltoncountyohio.gov/download.php?file=VoterListExport-20240912-no.csv>, although that URL is not expected to remain available. Fields named AddressPreDirectional, AddressNumber, AddressStreet, AddressSuffix, CityName, "OH", and AddressZip were pasted together before reducing the source to 2,000 unique registered-voter address examples. The fixture is not intended to be regenerated by package users.

Usage

```
voter_addresses()
```

Value

a character vector

Examples

```
voter_addresses() |>
  head()
```

zipcode_variant	Create ZIP code variants
-----------------	--------------------------

Description

An input ZIP code is used to generate variants (for, e.g., 45220):

- minus1: subtracting one from zipcode (45219)
- plus1: adding one to zipcode (45221)
- sub5: substituting the fifth digit of the ZIP code (45221, 45222, 45223, 45224, 45225, 45226, 45227, 45228, 45229)
- sub4: substituting the fourth digit of the ZIP code (45200, 45210, 45230, 45240, 45250, 45260, 45270, 45280, 45290)
- swap: swapping the second and third digits of the ZIP code (42520)

More than one variant type can be created at once and variants will be returned in the same order as they were requested (see examples).

Usage

```
zipcode_variant(x, variant = c("minus1", "plus1", "sub5", "sub4", "swap"))
```

Arguments

x	character length one; five digit ZIP code
variant	character one or more variants to create; see description

Value

character vector of five digit ZIP code variants

Examples

```
zipcode_variant("45220")

# order matters!
zipcode_variant("45220", c("minus1", "plus1"))
zipcode_variant("45220", c("plus1", "minus1"))

zipcode_variant("45220", "sub5")

zipcode_variant("45220", "swap")
```

Index

`addr (addr_number)`, [11](#)
`addr_fuzzy_left_join`, [3](#)
`addr_fuzzy_match`, [4](#)
`addr_left_join`, [6](#)
`addr_match`, [8](#)
`addr_match()`, [6](#)
`addr_match_prepare (addr_match)`, [8](#)
`addr_match_stage`, [10](#)
`addr_number`, [11](#)
`addr_place (addr_number)`, [11](#)
`addr_street (addr_number)`, [11](#)
`as_addr`, [14](#)

`clean_address_text`, [15](#)
`county_fips_lookup`, [16](#)

`elh_data`, [17](#)

`fuzzy_match`, [17](#)
`fuzzy_match_addr_field`
 (`addr_fuzzy_match`), [4](#)

`geocode`, [18](#)
`geocode()`, [22](#), [23](#)
`geocode_stage`, [22](#)
`geocode_table`, [23](#)
`geocode_zip (geocode)`, [18](#)

`match_addr_number`, [23](#)
`match_addr_street`, [24](#)
`match_zipcodes`, [28](#)

`nad`, [29](#)
`nad_catalog`, [32](#)
`nad_dataset`, [32](#)
`nad_download (nad)`, [29](#)
`nad_example_data`, [33](#)
`nad_install (nad)`, [29](#)
`nad_manifest`, [34](#)
`nad_manifest()`, [32](#)

`phonetic_street_key`, [34](#)

`taf`, [35](#)
`taf_catalog`, [36](#)
`taf_catalog()`, [38](#)
`taf_dataset`, [37](#)
`taf_ensure (taf_needed_counties)`, [39](#)
`taf_install (taf)`, [35](#)
`taf_manifest`, [38](#)
`taf_needed_counties`, [39](#)
`tag_usaddress`, [40](#)
`tiger_addr_feat`, [42](#)
`tiger_feat_names`, [42](#)

`voter_addresses`, [43](#)

`zipcode_variant`, [44](#)
`zipcode_variant()`, [20](#), [21](#)