Package ‘MazamaLocationUtils’

January 24, 2024

Type Package

Version 0.4.3

Title Manage Spatial Metadata for Known Locations

Maintainer Jonathan Callahan <jonathan.s.callahan@gmail.com>

Description Utility functions for discovering and managing metadata associated with spatially unique "known locations". Applications include all fields of environmental monitoring (e.g. air and water quality) where data are collected at stationary sites.

License GPL-3

URL https://github.com/MazamaScience/MazamaLocationUtils

BugReports https://github.com/MazamaScience/MazamaLocationUtils/issues

Depends R (>= 4.0)

Imports cluster, dplyr, geodist (>= 0.0.8), httr, jsonlite, leaflet, lubridate, magrittr, methods, MazamaCoreUtils (>= 0.5.1), MazamaSpatialUtils (>= 0.8.6), readr, rlang, stringr, tidygeocoder

Suggests knitr, markdown, testthat (>= 2.1.0), rmarkdown, roxygen2

Encoding UTF-8

VignetteBuilder knitr

LazyData true

RoxygenNote 7.2.3

NeedsCompilation no

Author Jonathan Callahan [aut, cre], Eli Grosman [ctb], Oliver Fogelin [ctb]

Repository CRAN

Date/Publication 2024-01-24 15:22:53 UTC
R topics documented:

APIKeys .......................................................... 3
clusterByDistance .............................................. 3
coreMetadataNames ........................................... 5
getLocationDataDir ........................................... 6
id_monitors_500 ................................................. 6
initializeMazamaSpatialUtils ................................. 7
LocationDataDir ................................................ 8
location_createID ............................................. 8
location_getCensusBlock ..................................... 9
location_getOpenCageInfo ................................... 10
location_getSingleAddressPhoton ......................... 12
location_getSingleAddressTexasAM ....................... 13
location_getSingleElevationUSGS ......................... 14
location_initialize .......................................... 15
MazamaLocationUtils ......................................... 17
or_monitors_500 .............................................. 18
setLocationDataDir .......................................... 19
table_addClustering .......................................... 20
table_addColumn ............................................. 21
table_addCoreMetadata ...................................... 22
table_addLocation ........................................... 24
table_addOpenCageInfo ...................................... 25
table_addSingleLocation .................................... 27
table_filterByDistance ...................................... 29
table_findAdjacentDistances ............................... 30
table_findAdjacentLocations ............................... 31
table_getDistanceFromTarget ............................... 32
table_getLocationID ......................................... 33
table_getNearestDistance .................................. 35
table_getNearestLocation .................................. 36
table_getRecordIndex ....................................... 37
table_initialize .............................................. 38
table_initializeExisting .................................... 39
table_leaflet .................................................. 41
table_leafletAdd ............................................. 42
table_load ..................................................... 43
table_removeColumn ......................................... 44
table_removeRecord ......................................... 45
table_save ..................................................... 46
table_updateColumn ......................................... 47
table_updateSingleRecord .................................. 49
validateLocationTbl .......................................... 50
validateMazamaSpatialUtils ................................. 50
wa_airfire_meta ............................................... 51
wa_monitors_500 ............................................. 52

Index ........................................................... 53
APIKeys

API keys for data services.

Description

This package maintains an internal set of API keys which users can set using setAPIKey(). These keys will be remembered for the duration of an R session. This functionality provides an abstraction layer in dependent packages so that data access functions can test for and access specific API keys with generic code.

Format

Character strings.

Details

The following functions help with the management of API keys:

getAPIKey() – Returns the API key associated with a web service. If provider == NULL a list is returned containing all recognized API keys.

setAPIKey() – Sets the API key associated with a web service. Silently returns previous value of the API key.

showAPIKeys() – Returns a list of all currently set API keys.

clusterByDistance

Add distance-clustering information to a dataframe

Description

Distance clustering is used to identify unique deployments of a sensor in an environmental monitoring field study. GPS-reported locations can be jittery and result in a sensor self-reporting from a cluster of nearby locations. Clustering helps resolve this by assigning a single location to the cluster.

Standard kmeans clustering does not work well when clusters can have widely differing numbers of members. A much better result is acheived with the Partitioning Around Medoids method available in cluster::pam().

The value of clusterDiameter is compared with the output of cluster::pam(...)$clusinfo[, 'av_diss'] to determine the number of clusters.
Usage

```
clusterByDistance(
  tbl,
  clusterDiameter = 1000,
  lonVar = "longitude",
  latVar = "latitude",
  maxClusters = 50
)
```

Arguments

- `tbl` Tibble with geolocation information.
- `clusterDiameter` Diameter in meters used to determine the number of clusters (see description).
- `lonVar` Name of longitude variable in the incoming tibble.
- `latVar` Name of the latitude variable in the incoming tibble.
- `maxClusters` Maximum number of clusters to try.

Value

Input tibble with additional columns: `clusterLon`, `clusterLat`, `clusterID`.

Note

In most applications, the `table_addClustering` function should be used as it implements two-stage clustering using `clusterByDistance()`.

References

When k-means clustering fails

See Also

- `table_removeRecord`

Examples

```r
library(MazamaLocationUtils)

# Fremont, Seattle 47.6504, -122.3509
# Magnolia, Seattle 47.6403, -122.3997
# Downtown Seattle 47.6055, -122.3370

fremont_x <- jitter(rep(-122.3509, 10), .0005)
fremont_y <- jitter(rep(47.6504, 10), .0005)
magnolia_x <- jitter(rep(-122.3997, 8), .0005)
magnolia_y <- jitter(rep(47.6403, 8), .0005)
```
downtown_x <- jitter(rep(-122.3370, 3), .0005)
downtown_y <- jitter(rep(47.6055, 3), .0005)

# Apply clustering
tbl <-
dplyr::tibble(
  longitude = c(fremont_x, magnolia_x, downtown_x),
  latitude = c(fremont_y, magnolia_y, downtown_y)
) %>%
clusterByDistance(
  clusterDiameter = 1000
)

plot(tbl$longitude, tbl$latitude, pch = tbl$clusterID)

---

coreMetadataNames  Names of standard spatial metadata columns

**Description**

Character string identifiers of the minimum set of fields required for a table to be considered a valid "known locations" table.

coreMetadataNames <- c(
  "locationID", # from MazamaLocationUtils::location_createID()
  "locationName", # from MazamaLocationUtils::location_initialize()
  "longitude", # user supplied
  "latitude", # user supplied
  "elevation", # from MazamaLocationUtils::getSingleElevation_USGS()
  "countryCode", # from MazamaSpatialUtils::getCountryCode()
  "stateCode", # from MazamaSpatialUtils::getStateCode()
  "countyName", # from MazamaSpatialUtils::getUSCounty()
  "timezone", # from MazamaSpatialUtils::getTimezone()
  "houseNumber", # from MazamaLocationUtils::getSingleAddress_Photon()
  "street", # from MazamaLocationUtils::getSingleAddress_Photon()
  "city", # from MazamaLocationUtils::getSingleAddress_Photon()
  "postalCode" # from MazamaLocationUtils::getSingleAddress_Photon()
)

**Usage**

coreMetadataNames

**Format**

A vector with 3 elements

**Details**

coreMetadataNames
getLocationDataDir  Get location data directory

Description
Returns the directory path where known location data tables are located.

Usage
getLocationDataDir()

Value
Absolute path string.

See Also
LocationDataDir
setLocationDataDir

id_monitors_500  Idaho monitor locations dataset

Description
The id_monitors_500 dataset provides a set of known locations associated with Idaho state air quality monitors. This dataset was generated on 2023-10-24 by running:

library(AirMonitor)
library(MazamaLocationUtils)
initializeMazamaSpatialUtils()
setLocationDataDir("./data")

monitor <- monitor_loadLatest() %>% monitor_filter(stateCode == "ID")
lons <- monitor$meta$longitude
lats <- monitor$meta$latitude

table_initialize() %>%
table_addLocation(
lons, lats,
distanceThreshold = 500,
elevationService = "usgs",
addressService = "photon"
) %>%
table_save("id_monitors_500")
Initialize MazamaSpatialUtils package

Usage

initializeMazamaSpatialUtils(spatialDataDir = "~/Data/Spatial")

Arguments

spatialDataDir  Directory where MazamaSpatialUtils datasets are found.

Examples

library(MazamaLocationUtils)

# Fail gracefully if any resources are not available
try({
  
  # Set up directory for spatial data
  spatialDataDir <- tempdir()  # typically "~/Data/Spatial"

  initializeMazamaSpatialUtils(spatialDataDir = spatialDataDir)
})
MazamaSpatialUtils::setSpatialDataDir(spatialDataDir)

exists("NaturalEarthAdm1")
initializeMazamaSpatialUtils(spatialDataDir)
exists("NaturalEarthAdm1")
class(NaturalEarthAdm1)

}, silent = FALSE)

---

**LocationDataDir**

*Directory for location data*

---

**Description**

This package maintains an internal directory path which users can set using `setLocationDataDir()`. All package functions use this directory whenever known location tables are accessed. The default setting when the package is loaded is `getwd()`.

**Format**

Absolute path string.

**See Also**

`getLocationDataDir`

`setLocationDataDir`

---

**location_createID**

*Create one or more unique locationIDs*

---

**Description**

A unique locationID is created for each incoming longitude and latitude. See `MazamaCoreUtils::createLocationID` for details.

At `precision = 10`, this results in a maximum error of 0.6 meters which is more than precise enough for environmental monitoring studies making use of this package.

An excellent way to become familiar with geohash is through the GeoHash Explorer.

**Usage**

```r
location_createID(
  longitude = NULL,
  latitude = NULL,
  algorithm = c("geohash", "digest"),
  precision = 10
)
```
location_getCensusBlock

Arguments

longitude Vector of longitudes in decimal degrees E.
latitude Vector of latitudes in decimal degrees N.
algorithm Algorithm to use – either "geohash" or "digest".
precision precision argument used when encoding with "geohash".

Value

Vector of character locationIDs.

Note

The "digest" algorithm is deprecated but provided for backwards compatibility with databases that were built using locationIDs generated with this algorithm.

References

https://en.wikipedia.org/wiki/Decimal_degrees
https://www.johndcook.com/blog/2017/01/10/probability-of-secure-hash-collisions/

Examples

library(MazamaLocationUtils)

# Wenatchee
lon <- -120.325278
lat <- 47.423333
locationID <- location_createID(lon, lat)
print(locationID)

location_createID(lon, lat, algorithm = "geohash")
location_createID(lon, lat, algorithm = "geohash", precision = 7)

location_getCensusBlock

_Get census block data from the FCC API_

Description

The FCC Block API is used get census block, county, and state FIPS associated with the longitude and latitude. The following list of data is returned:

- stateCode
- countyName
- censusBlock

The data from this function should be considered to be the gold standard for state and county. i.e. this information could and should be used to override information we get elsewhere.
location_getOpenCageInfo

Get location information from OpenCage

Usage

location_getOpenCageInfo

Arguments

longitude Single longitude in decimal degrees E.
latitude Single latitude in decimal degrees N.
censusYear Year the census was taken.
verbose Logical controlling the generation of progress messages.

Value

List of census block/county/state data.

References

https://geo.fcc.gov/api/census/

Examples

library(MazamaLocationUtils)
# Fail gracefully if any resources are not available
try({
  # Wenatchee
  lon <- -120.325278
  lat <- 47.423333

  censusList <- location_getOpenCageInfo(lon, lat)
  str(censusList)
}, silent = FALSE)
Description

The OpenCage reverse geocoding service is used to obtain all available information for a specific location.

The data from OpenCage should be considered to be the gold standard for address information and should be used to override information we get elsewhere.

Usage

location_getOpenCageInfo(longitude = NULL, latitude = NULL, verbose = FALSE)

Arguments

longitude Single longitude in decimal degrees E.
latitude Single latitude in decimal degrees N.
verbose Logical controlling the generation of progress messages.

Value

Single-row tibble with OpenCage information.

Note

The OpenCage service requires an API key which can be obtained from their web site. This API key must be set as an environment variable with:

Sys.setenv("OPENCAGE_KEY" = "YOUR_PERSONAL_API_KEY")

The OpenCage "free trial" level allows for 1 request/sec and a maximum of 2500 requests per day.

References

https://opencagedata.com

Examples

library(MazamaLocationUtils)

# Fail gracefully if any resources are not available
try{

  # Wenatchee
  lon <- -120.325278
  lat <- 47.423333

  Sys.setenv("OPENCAGE_KEY" = "YOUR_PERSONAL_API_KEY")

  openCageTbl <- location_getOpenCageInfo(lon, lat)
  dplyr::glimpse(openCageTbl)
Description

The Photon API is used get address data associated with the longitude and latitude. The following list of data is returned:

- houseNumber
- street
- city
- stateCode
- stateName
- postalCode
- countryCode
- countryName

The function makes an effort to convert both state and country Name into Code with codes defaulting to NA. Both Name and Code are returned so that improvements can be made in the conversion algorithm.

Usage

```r
location_getSingleAddress_Photon(
  longitude = NULL,
  latitude = NULL,
  baseUrl = "https://photon.komoot.io/reverse",
  verbose = TRUE
)
```

Arguments

- **longitude**: Single longitude in decimal degrees E.
- **latitude**: Single latitude in decimal degrees N.
- **baseUrl**: Base URL for data queries.
- **verbose**: Logical controlling the generation of progress messages.

Value

List of address components.
**location_getSingleAddress_TexasAM**

**References**

https://photon.komoot.io

**Examples**

```r
library(MazamaLocationUtils)

# Fail gracefully if any resources are not available
try({

  # Set up standard directories and spatial data
  spatialDataDir <- tempdir() # typically "~/Data/Spatial"
  initializeMazamaSpatialUtils(spatialDataDir)

  # Wenatchee
  lon <- -120.325278
  lat <- 47.423333

  addressList <- location_getSingleAddress_Photon(lon, lat)
  str(addressList)
}, silent = FALSE)
```

---

**location_getSingleAddress_TexasAM**

*Get an address from the Texas A&M reverse geocoding service*

**Description**

Texas A&M APIs are used to determine the address associated with the longitude and latitude.

**Usage**

```r
location_getSingleAddress_TexasAM(
  longitude = NULL,
  latitude = NULL,
  apiKey = NULL,
  verbose = TRUE
)
```

**Arguments**

- `longitude` Single longitude in decimal degrees E.
- `latitude` Single latitude in decimal degrees N.
- `apiKey` Texas A&M Geocoding requires an API key. The first 2500 requests are free.
- `verbose` Logical controlling the generation of progress messages.
Value

Numeric elevation value.

References

https://geoservices.tamu.edu/Services/ReverseGeocoding/WebService/v04_01/HTTP.aspx

Examples

```r
## Not run:
library(MazamaLocationUtils)

# Fail gracefully if any resources are not available
try({

  # Wenatchee
  longitude <- -122.47
  latitude <- 47.47
  apiKey <- YOUR_PERSONAL_API_KEY

  location_getSingleAddress_TexasAM(longitude, latitude, apiKey)
}, silent = FALSE)

## End(Not run)
```

location_getSingleElevation_USGS

Get elevation data from a USGS web service

Description

USGS APIs are used to determine the elevation in meters associated with the longitude and latitude.

Note: The conversion factor for meters to feet is 3.28084.

Usage

```r
location_getSingleElevation_USGS(
  longitude = NULL,
  latitude = NULL,
  verbose = TRUE
)
```
location_initialize

Arguments

longitude  Single longitude in decimal degrees E.
latitude   Single latitude in decimal degrees N.
verbose    Logical controlling the generation of progress messages.

Value

Numeric elevation value.

References

https://epqs.nationalmap.gov/v1/docs

Examples

library(MazamaLocationUtils)

# Fail gracefully if any resources are not available
try({
  # Wenatchee
  longitude <- -120.325278
  latitude <- 47.423333

  location_getSingleElevation_USGS(longitude, latitude)
}, silent = FALSE)

location_initialize  Create known location record with core metadata

Description

Creates a known location record with the following columns of core metadata:

• locationID
• locationName
• longitude
• latitude
• elevation
• countryCode
• stateCode
• countyName
location_initialize

- timezone
- houseNumber
- street
- city
- postalCode

Usage

```r
location_initialize(
  longitude = NULL,
  latitude = NULL,
  stateDataset = "NaturalEarthAdm1",
  elevationService = NULL,
  addressService = NULL,
  precision = 10,
  verbose = TRUE
)
```

Arguments

- **longitude**: Single longitude in decimal degrees E.
- **latitude**: Single latitude in decimal degrees N.
- **stateDataset**: Name of spatial dataset to use for determining state
- **elevationService**: Name of the elevation service to use for determining the elevation. Default: NULL skips this step. Accepted values: "usgs".
- **addressService**: Name of the address service to use for determining the street address. Default: NULL skips this step. Accepted values: "photon".
- **precision**: precision argument passed on to location_createID.
- **verbose**: Logical controlling the generation of progress messages.

Value

Tibble with a single new known location.

Examples

```r
library(MazamaLocationUtils)

# Fail gracefully if any resources are not available
try{

  # Set up standard directories and spatial data
  spatialDataDir <- tempdir() # typically "~/Data/Spatial"
  initializeMazamaSpatialUtils(spatialDataDir)
```
# Wenatchee
lon <- -120.325278
lat <- 47.423333

locationRecord <- location_initialize(lon, lat)
str(locationRecord)

MazamaLocationUtils

Manage Spatial Metadata for Known Locations

Description
A suite of utility functions for discovering and managing metadata associated with sets of spatially unique "known locations".

This package is intended to be used in support of data management activities associated with fixed locations in space. The motivating fields include both air and water quality monitoring where fixed sensors report at regular time intervals.

Details
When working with environmental monitoring time series, one of the first things you have to do is create unique identifiers for each individual time series. In an ideal world, each environmental time series would have both a locationID and a deviceID that uniquely identify the specific instrument making measurements and the physical location where measurements are made. A unique timeseriesID could be produced as locationID_deviceID. Metadata associated with each timeseriesID would contain basic information needed for downstream analysis including at least:

timeseriesID, locationID, deviceID, longitude, latitude, ...

- An extended time series for an occasionally re-positioned sensor would group by deviceID.
- Multiple sensors placed at a single location could be be grouped by locationID.
- Maps would be created using longitude, latitude.
- Time series would be accessed from a secondary data table with timeseriesID.

Unfortunately, we are rarely supplied with a truly unique and truly spatial locationID. Instead we often use deviceID or an associated non-spatial identifier as a stand-in for locationID. Complications we have seen include:

- GPS-reported longitude and latitude can have jitter in the fourth or fifth decimal place making it challenging to use them to create a unique locationID.
- Sensors are sometimes re-positioned in what the scientist considers the "same location".
• Data for a single sensor goes through different processing pipelines using different identifiers and is later brought together as two separate time series.
• The spatial scale of what constitutes a "single location" depends on the instrumentation and scientific question being asked.
• Deriving location-based metadata from spatial datasets is computationally intensive unless saved and identified with a unique locationID.
• Automated searches for spatial metadata occasionally produce incorrect results because of the non-infinite resolution of spatial datasets.

This package attempts to address all of these issues by maintaining a table of known locations for which CPU intensive spatial data calculations have already been performed. While requests to add new locations to the table may take some time, searches for spatial metadata associated with existing locations are simple lookups.

Working in this manner will solve the problems initially mentioned but also provides further useful functionality.

• Administrators can correct entries in the collectionName table. (e.g. locations in river bends that even high resolution spatial datasets mis-assign)
• Additional, non-automatable metadata can be added to collectionName. (e.g. commonly used location names within a community of practice)
• Different field campaigns can have separate collectionName tables.
• .csv or .rda versions of well populated tables can be downloaded from a URL and used locally, giving scientists working with known locations instant access to spatial data that otherwise requires special skills, large datasets and lots of compute cycles.

---

or_monitors_500  Oregon monitor locations dataset

Description

The or_monitors_500 dataset provides a set of known locations associated with Oregon state air quality monitors. This dataset was generated on 2023-10-24 by running:

```r
library(AirMonitor)
library(MazamaLocationUtils)
initializeMazamaSpatialUtils()
setLocationDataDir("./data")
monitor <- monitor_loadLatest() %>% monitor_filter(stateCode == "OR")
lons <- monitor$meta$longitude
lats <- monitor$meta$latitude

table_initialize() %>%
table_addLocation(
```
```r
lons, lats,
distanceThreshold = 500,
elevationService = "usgs",
addressService = "photon"
}%>
%>
table_save("or_monitors_500")
```

**Usage**

`or_monitors_500`

**Format**

A tibble with 64 rows and 13 columns of data.

**See Also**

`id_monitors_500`

`wa_monitors_500`

---

### setLocationDataDir

**Set location data directory**

**Description**

Sets the data directory where known location data tables are located. If the directory does not exist, it will be created.

**Usage**

`setLocationDataDir(dataDir)`

**Arguments**

- `dataDir` Directory where location tables are stored.

**Value**

Silently returns previous value of the data directory.

**See Also**

`LocationDataDir`

`getLocationDataDir`
table_addClustering

Add clustering information to a dataframe

Description

Clustering is used to identify unique deployments of a sensor in an environmental monitoring field study.

Sensors will be moved around from time to time, sometimes across the country and sometimes across the street. We would like to assign unique identifiers to each new "deployment" but not when the sensor is moved a short distance.

We use clustering to find an appropriate number of unique "deployments". The sensitivity of this algorithm can be adjusted with the clusterDiameter argument.

Standard kmeans clustering does not work well when clusters can have widely differing numbers of members. A much better result is achieved with the Partitioning Around Medoids method available in cluster::pam().

The value of clusterRadius is compared with the output of cluster::pam(...)$clusinfo[, 'av_diss'] to determine the number of clusters.

Usage

```
    table_addClustering(
      tbl,
      clusterDiameter = 1000,
      lonVar = "longitude",
      latVar = "latitude",
      maxClusters = 50
    )
```

Arguments

- `tbl` Tibble with geolocation information (e.g.,
- `clusterDiameter` Diameter in meters used to determine the number of clusters (see description).
- `lonVar` Name of longitude variable in the incoming tibble.
- `latVar` Name of the latitude variable in the incoming tibble.
- `maxClusters` Maximum number of clusters to try.

Value

Input tibble with additional columns: `clusterLon`, `clusterLat`.

Note

The `table_addClustering()` function implements two-stage clustering using `clusterByDistance`. If the first attempt at clustering produces clustered locations that are still too close to each other, another round of clustering is performed using the results of the previous attempt. This two-stage approach seems to work well in practice.
References

When k-means clustering fails

See Also

clusterByDistance

Examples

library(MazamaLocationUtils)

# Fremont, Seattle 47.6504, -122.3509
# Magnolia, Seattle 47.6403, -122.3997
# Downtown Seattle 47.6055, -122.3370

fremont_x <- jitter(rep(-122.3509, 10), .0005)
fremont_y <- jitter(rep(47.6504, 10), .0005)
magnolia_x <- jitter(rep(-122.3997, 8), .0005)
magnolia_y <- jitter(rep(47.6403, 8), .0005)
downtown_x <- jitter(rep(-122.3370, 3), .0005)
downtown_y <- jitter(rep(47.6055, 3), .0005)

# Apply clustering
tbl <-
dplyr::tibble(
  longitude = c(fremont_x, magnolia_x, downtown_x),
  latitude = c(fremont_y, magnolia_y, downtown_y)
) %>%
  table_addClustering(
    clusterDiameter = 1000
  )

plot(tbl$longitude, tbl$latitude, pch = tbl$clusterID)
table_addCoreMetadata

locationID = NULL,
locationData = NULL,
verbose = TRUE
)

Arguments

locationTbl Tibble of known locations.
columnName Name to use for the new column.
locationID Vector of locationID strings.
locationData Vector of data to used at matching records.
verbose Logical controlling the generation of progress messages.

Value

Updated tibble of known locations.

See Also

table_removeColumn
table_updateColumn

Examples

library(MazamaLocationUtils)

# Starting table
locationTbl <- get(data("wa_monitors_500"))
names(locationTbl)

# Add an empty column
locationTbl <-
  locationTbl %>%
  table_addColumn("AQSID")
names(locationTbl)
Description

An existing table will be amended to guarantee that it includes the following core metadata columns.

- locationID
- locationName
- longitude
- latitude
- elevation
- countryCode
- stateCode
- countyName
- timezone
- houseNumber
- street
- city
- postalCode

The longitude and latitude columns are required to exist in the incoming tibble but all others are optional.

If any of these core metadata columns are found, they will be retained.

The locationID will be generated (anew if already found) from existing longitude and latitude data.

Other core metadata columns will be filled with NA values of the proper type.

The result is a tibble with all of the core metadata columns. Theses columns must then be filled in to create a usable "known locations" table.

Usage

table_addCoreMetadata(locationTbl = NULL, precision = 10)

Arguments

- locationTbl: Tibble of known locations. This input tibble need not be a standardized "known location" with all required columns. They will be added.
- precision: precision argument passed on to location_createID.

Value

Tibble with the metadata columns required in a "known locations" table.

Note

No check is performed for overlapping locations. The returned tibble has the structure of a "known locations" table and is a good starting place for investigation. But further work is required to produce a valid table of "known locations" associated with a specific spatial scale.
table_addLocation

Add new known location records to a table

Description

Incoming longitude and latitude values are compared against the incoming locationTbl to see if they are already within distanceThreshold meters of an existing entry. A new record is created for each location that is not already found in locationTbl.

Usage

```r
table_addLocation(
  locationTbl = NULL,
  longitude = NULL,
  latitude = NULL,
  distanceThreshold = NULL,
  stateDataset = "NaturalEarthAdm1",
  elevationService = NULL,
  addressService = NULL,
  verbose = TRUE
)
```

Arguments

- `locationTbl` Tibble of known locations.
- `longitude` Vector of longitudes in decimal degrees E.
- `latitude` Vector of latitudes in decimal degrees N.
- `distanceThreshold` Distance in meters.
- `stateDataset` Name of spatial dataset to use for determining state codes, Default: 'NaturalEarthAdm1'
- `elevationService` Name of the elevation service to use for determining the elevation. Default: NULL skips this step. Accepted values: "usgs".
- `addressService` Name of the address service to use for determining the street address. Default: NULL skips this step. Accepted values: "photon".
- `verbose` Logical controlling the generation of progress messages.

Value

Updated tibble of known locations.

Note

This function is a vectorized version of table_addSingleLocation().
See Also

- `table_addSingleLocation`
- `table_removeRecord`
- `table_updateSingleRecord`

Examples

```r
library(MazamaLocationUtils)

# Fail gracefully if any resources are not available
try({
  # Set up standard directories and spatial data
  spatialDataDir <- tempdir() # typically "~/Data/Spatial"
  initializeMazamaSpatialUtils(spatialDataDir)

  locationTbl <- get(data("wa_monitors_500"))

  # Coulee City, WA
  lon <- -119.290904
  lat <- 47.611942

  locationTbl <-
    locationTbl %>%
      table_addLocation(lon, lat, distanceThreshold = 500)

dplyr::glimpse(locationTbl)
}, silent = FALSE)
```

---

### table_addOpenCageInfo

*Add address fields to a known location table*

#### Description

The OpenCage reverse geocoding service is used to update an existing table. Updated columns include:

- countryCode
- stateCode
- countyName
- timezone
- houseNumber
- street
• city
• postalCode
• address

When `replaceExisting = TRUE`, all existing address fields are discarded in favor of the OpenCage versions. To only fill in missing values in `locationTbl`, use `replaceExisting = FALSE`.

The OpenCage service returns a large number of fields, some of which may be useful. To add all OpenCage fields to a location table, use `retainOpenCage = TRUE`. This will append 78+ fields of information, each each named with a prefix of "opencage_".

Usage

```r
table_addOpenCageInfo(
  locationTbl = NULL,
  replaceExisting = FALSE,
  retainOpenCage = FALSE,
  verbose = FALSE
)
```

Arguments

- `locationTbl` Tibble of known locations.
- `replaceExisting` Logical specifying whether to replace existing data with data obtained from OpenCage.
- `retainOpenCage` Logical specifying whether to retain all fields obtained from OpenCage, each named with a prefix of opencage_.
- `verbose` Logical controlling the generation of progress messages.

Value

Tibble of "known locations" enhanced with information from the OpenCage reverse geocoding service.

Note

The OpenCage service requires an API key which can be obtained from their web site. This API key must be set as an environment variable with:

```r
Sys.setenv("OPENCAGE_KEY" = "<your api key>")
```

Parameters are set for use at the OpenCage "free trial" level which allows for 1 request/sec and a maximum of 2500 requests per day.

Because of the 1 request/sec default, it is recommended that `table_addOpenCageInfo()` only be used in an interactive session when updating a table with a large number of records.

References

https://opencagedata.com
Examples

library(MazamaLocationUtils)

# Fail gracefully if any resources are not available
try({
  myTbl <- id_monitors_500[1:3,]
  myTbl$countryCode[1] <- NA
  myTbl$countryCode[2] <- "WRONG"
  myTbl$countyName[3] <- "WRONG"
  myTbl$timeZone <- NA

dplyr::glimpse(myTbl)

Sys.setenv("OPENCAGE_KEY" = "<YOUR_KEY>"

  table_addOpenCageInfo(myTbl) %>%
    dplyr::glimpse()

  table_addOpenCageInfo(myTbl, replaceExisting = TRUE) %>%
    dplyr::glimpse()

  table_addOpenCageInfo(myTbl, replaceExisting = TRUE, retainOpenCage = TRUE) %>%
    dplyr::glimpse()
}, silent = FALSE)

---

**table_addSingleLocation**

Add a single new known location record to a table

Description

Incoming longitude and latitude values are compared against the incoming `locationTbl` to see if they are already within `distanceThreshold` meters of an existing entry. A new record is created for if the location is not already found in `locationTbl`.

Usage

```r
  table_addSingleLocation(
    locationTbl = NULL,
    longitude = NULL,
    latitude = NULL,
    distanceThreshold = NULL,
    stateDataset = "NaturalEarthAdm1",
    elevationService = NULL,
    addressService = NULL,
  )
```
Arguments

locationTbl  Tibble of known locations.
longitude     Single longitude in decimal degrees E.
latitude      Single latitude in decimal degrees N.
distanceThreshold  Distance in meters.
stateDataset  Name of spatial dataset to use for determining state codes, Default: "NaturalEarthAdm1".
elevationService  Name of the elevation service to use for determining the elevation. Default: NULL. Accepted values: "usgs".
addressService Name of the address service to use for determining the street address. Default: NULL. Accepted values: "photon".
verbose       Logical controlling the generation of progress messages.

Value

Updated tibble of known locations.

See Also

table_addLocation
table_removeRecord
table_updateSingleRecord

Examples

library(MazamaLocationUtils)

# Fail gracefully if any resources are not available
try({
  # Set up standard directories and spatial data
  spatialDataDir <- tempdir() # typically "/Data/Spatial"
  initializeMazamaSpatialUtils(spatialDataDir)

  locationTbl <- get(data("wa_monitors_500"))
  nrow(locationTbl)

  # Coulee City, WA
  lon <- -119.290904
  lat <- 47.611942
})
locationTbl <-
  locationTbl %>%
  table_addSingleLocation(lon, lat, distanceThreshold = 500)

nrow(locationTbl)

}, silent = FALSE)

---

**table_filterByDistance**

*Return known locations near a target location*

**Description**

Returns a tibble of the known locations from locationTbl that are within distanceThreshold meters of the target location specified by longitude and latitude.

**Usage**

```r
table_filterByDistance(
  locationTbl = NULL,
  longitude = NULL,
  latitude = NULL,
  distanceThreshold = NULL,
  measure = c("geodesic", "haversine", "vincenty", "cheap")
)
```

**Arguments**

- `locationTbl` Tibble of known locations.
- `longitude` Target longitude in decimal degrees E.
- `latitude` Target latitude in decimal degrees N.
- `distanceThreshold` Distance in meters.
- `measure` One of "haversine" "vincenty", "geodesic", or "cheap" specifying desired method of geodesic distance calculation.

**Value**

Tibble of known locations.

**Note**

Only a single target location is allowed.
Examples

library(MazamaLocationUtils)

locationTbl <- get(data("wa_monitors_500"))

# Too small a distanceThreshold will not find a match
locationTbl %>%
  table_filterByDistance(
    longitude = -117.3647,
    latitude = 47.6725,
    distanceThreshold = 10
  ) %>%
  dplyr::glimpse()

# Expanding the distanceThreshold will find several
locationTbl %>%
  table_filterByDistance(
    longitude = -117.3647,
    latitude = 47.6725,
    distanceThreshold = 10000
  ) %>%
  dplyr::glimpse()

table_findAdjacentDistances

Find distances between adjacent locations in a known locations table

Description

Calculate distances between all locations within a known locations table and return a tibble with the row indices and separation distances of those records separated by less than distanceThreshold meters. Records are returned in order of distance.

It is useful when working with new metadata tables to identify adjacent locations early on so that decisions can be made about the appropriateness of the specified distanceThreshold.

Usage

table_findAdjacentDistances(
  locationTbl = NULL,
  distanceThreshold = NULL,
  measure = c("geodesic", "haversine", "vincenty", "cheap")
)

Arguments

locationTbl Tibble of known locations.
distanceThreshold Distance in meters.
measure

One of "haversine", "vincenty", "geodesic", or "cheap" specifying desired method of geodesic distance calculation. See geodist::geodist for details.

Value

Tibble of row indices and distances for those locations separated by less than distanceThreshold meters.

Note

The measure "cheap" may be used to speed things up depending on the spatial scale being considered. Distances calculated with measure = "cheap" will vary by a few meters compared with those calculated using measure = "geodesic".

Examples

```r
library(MazamaLocationUtils)

meta <- wa_airfire_meta

# Any locations closer than 2 km?
table_findAdjacentDistances(meta, distanceThreshold = 2000)

# How about 4 km?
table_findAdjacentDistances(meta, distanceThreshold = 4000)
```

Description

Calculate distances between all locations within a known locations table and return a tibble containing all records that have an adjacent location separated by less than distanceThreshold meters. The return tibble is ordered by separation distance.

It is useful when working with new metadata tables to identify adjacent locations early on so that decisions can be made about the appropriateness of the specified distanceThreshold.

Usage

```r
table_findAdjacentLocations(
  locationTbl = NULL,
  distanceThreshold = NULL,
  measure = c("geodesic", "haversine", "vincenty", "cheap")
)
```
table_getDistanceFromTarget

Return distances and directions from a target location to known locations

Description

Returns a tibble with the same number of rows as locationTbl containing the distance and direction from the target location specified by longitude and latitude to each known location found in locationTbl.
### table_getLocationID

**Return IDs of known locations**

**Description**

Returns a vector of `locationIDs` for the known locations that each incoming location will be assigned to within the given. If more than one known location exists within the given `distanceThreshold`, the closest will be assigned. `NA` will be returned for each incoming that cannot be assigned to a known location in `locationTbl`.

**Usage**

```r
library(MazamaLocationUtils)

locationTbl <- get(data("wa_monitors_500"))

locationTbl %>%
  table_getDistanceFromTarget(
    longitude = -117.3647,
    latitude = 47.6725
  ) %>%
dplyr::glimpse()
```

**Argument**

- `locationTbl`: Tibble of known locations.
- `longitude`: Target longitude in decimal degrees E.
- `latitude`: Target latitude in decimal degrees N.
- `measure`: One of "geodesic", "haversine", "vincenty" or "cheap" specifying desired method of geodesic distance calculation.

**Value**

Tibble of distances in meters and cardinal directions from a target location.

**Note**

Only a single target location is allowed.

**Examples**

```r
library(MazamaLocationUtils)

locationTbl <- get(data("wa_monitors_500"))

locationTbl %>%
  table_getDistanceFromTarget(
    longitude = -117.3647,
    latitude = 47.6725
  ) %>%
dplyr::glimpse()
```
Usage

```r
table_getLocationID(
  locationTbl = NULL,
  longitude = NULL,
  latitude = NULL,
  distanceThreshold = NULL,
  measure = c("geodesic", "haversine", "vincenty", "cheap")
)
```

Arguments

- `locationTbl`: Tibble of known locations.
- `longitude`: Vector of longitudes in decimal degrees E.
- `latitude`: Vector of latitudes in decimal degrees N.
- `distanceThreshold`: Distance in meters.
- `measure`: One of "geodesic", "haversine", "vincenty" or "cheap" specifying desired method of geodesic distance calculation. See `geodist`.

Value

Vector of known locationIDs.

Note

The measure "cheap" may be used to speed things up depending on the spatial scale being considered. Distances calculated with `measure = "cheap"` will vary by a few meters compared with those calculated using `measure = "geodesic"`.

Examples

```r
locationTbl <- get(data("wa_monitors_500"))

# Wenatchee
lon <- -120.325278
lat <- 47.423333

# Too small a distanceThreshold will not find a match
table_getLocationID(locationTbl, lon, lat, distanceThreshold = 50)

# Expanding the distanceThreshold will find one
table_getLocationID(locationTbl, lon, lat, distanceThreshold = 5000)
```
table_getNearestDistance

Return distances to nearest known locations

Description

Returns distances between target locations and the closest location found in locationTbl (if any). Target locations are specified with longitude and latitude.

For each target location, only a single distance to the closest known location is returned. If no known location is found within distanceThreshold, the distance associated with that target location will be NA. The length and order of resulting distances will match the order of the incoming target locations.

Usage

```r
table_getNearestDistance(
  locationTbl = NULL,
  longitude = NULL,
  latitude = NULL,
  distanceThreshold = NULL,
  measure = c("geodesic", "haversine", "vincenty", "cheap")
)
```

Arguments

- **locationTbl**: Tibble of known locations.
- **longitude**: Vector of target longitudes in decimal degrees E.
- **latitude**: Vector of target latitudes in decimal degrees N.
- **distanceThreshold**: Distance in meters.
- **measure**: One of "geodesic", "haversine", "vincenty" or "cheap" specifying desired method of geodesic distance calculation.

Value

Vector of closest distances between target locations and known locations.

Use Case

You may have a set of locations of interest for which you want to assess whether any monitoring locations are nearby. In this case, the locations of interest will provide longitude and latitude while locationTbl will be the known location table associated with the monitoring locations.

The resulting vector of distances will tell you the distance, for each target location, to the nearest monitoring location.
Note

The measure "cheap" may be used to speed things up depending on the spatial scale being considered. Distances calculated with measure = "cheap" will vary by a few meters compared with those calculated using measure = "geodesic".

See geodist for details.

Examples

library(MazamaLocationUtils)

locationTbl <- get(data("wa_monitors_500"))

# Wenatchee
lon <- -120.325278
lat <- 47.423333

# Too small a distanceThreshold will not find a match
table_getNearestDistance(locationTbl, lon, lat, distanceThreshold = 50)

# Expanding the distanceThreshold will find one
table_getNearestDistance(locationTbl, lon, lat, distanceThreshold = 5000)

Description

Returns a tibble of the known locations from locationTbl that are closest to the vector of target locations specified by longitude and latitude. Only a single known location is returned for each incoming target location. If no known location is found for a particular incoming location, that record in the tibble will contain all NA.

Usage

table_getNearestLocation(
    locationTbl = NULL,
    longitude = NULL,
    latitude = NULL,
    distanceThreshold = NULL
    )

Arguments

locationTbl Tibble of known locations.
longitude Vector of longitudes in decimal degrees E.
table_getRecordIndex

Value

Tibble of known locations.

Examples

library(MazamaLocationUtils)

locationTbl <- get(data("wa_monitors_500"))

# Wenatchee
lon <- -120.325278
lat <- 47.423333

# Too small a distanceThreshold will not find a match
table_getNearestLocation(locationTbl, lon, lat, distanceThreshold = 50) %>% str()

# Expanding the distanceThreshold will find one
table_getNearestLocation(locationTbl, lon, lat, distanceThreshold = 5000) %>% str()

Description

Returns a vector of locationTbl row indexes for the locations associated with each locationID.

Usage

table_getRecordIndex(locationTbl = NULL, locationID = NULL, verbose = TRUE)

Arguments

locationTbl Tibble of known locations.
locationID Vector of locationID strings.
verbose Logical controlling the generation of progress messages.

Value

Vector of locationTbl row indexes.
Examples

```r
library(MazamaLocationUtils)

locationTbl <- get(data("wa_monitors_500"))

# Wenatchee
lon <- -120.325278
lat <- 47.423333

# Get the locationID first
locationID <- table_getLocationID(locationTbl, lon, lat, distanceThreshold = 5000)

# Now find the row associated with this ID
recordIndex <- table_getRecordIndex(locationTbl, locationID)

str(locationTbl[recordIndex,])
```

---

### `table_initialize`

Create an empty known location table

**Description**

Creates an empty known location tibble with the following columns of core metadata:

- locationID
- locationName
- longitude
- latitude
- elevation
- countryCode
- stateCode
- countyName
- timezone
- houseNumber
- street
- city
- postalCode

**Usage**

```r
table_initialize()
```
**Value**

Empty known location tibble with the specified metadata columns.

**Examples**

```r
library(MazamaLocationUtils)

# Create an empty Tbl
dpyr::glimpse(emptyTbl)
```

---

**Description**

An existing table may have much of the data that is needed for a known location table. This function accepts an incoming table and searches for required columns:

- locationID
- locationName
- longitude
- latitude
- elevation
- countryCode
- stateCode
- countyName
- timezone
- houseNumber
- street
- city
- postalCode

The longitude and latitude columns are required but all others are optional. If any of these optional columns are found, they will be used and the often slow and sometimes slightly inaccurate steps to generate that information will be skipped for locations that have non-missing data. Any additional columns of information that are not part of the required core metadata will be retained.

This method skips the assignment of columns like elevation and all address related fields that require web service requests.

Compared to initializing a brand new table and populating it one record at a time, this is a much faster way of creating a known location table from a pre-existing table of metadata.
Usage

table_initializeExisting(
  locationTbl = NULL,
  stateDataset = "NaturalEarthAdm1",
  countryCodes = NULL,
  distanceThreshold = NULL,
  measure = c("geodesic", "haversine", "vincenty", "cheap"),
  precision = 10,
  verbose = TRUE
)

Arguments

locationTbl Tibble of known locations. This input tibble need not be a standardized "known location" table with all required columns. Missing columns will be added.

stateDataset Name of spatial dataset to use for determining state codes. Default: 'NaturalEarthAdm1'

countryCodes Vector of country codes used to optimize spatial searching. (See ?MazamaSpatialUtils::getStateCode())

distanceThreshold Distance in meters.

measure One of "haversine", "vincenty", "geodesic", or "cheap" specifying desired method of geodesic distance calculation. See ?geodist::geodist.

precision precision argument passed on to location_createID.

verbose Logical controlling the generation of progress messages.

Value

Known location tibble with the specified metadata columns. Any locations whose circles (as defined by distanceThreshold) overlap will generate warning messages.

It is incumbent upon the user to address overlapping locations by one of:

1. reduce the distanceThreshold until no overlaps occur
2. assign one of the overlapping locations to the other location

Note

The measure "cheap" may be used to speed things up depending on the spatial scale being considered. Distances calculated with measure = "cheap" will vary by a few meters compared with those calculated using measure = "geodesic".
table_leaflet | Leaflet interactive map for known locations

Description

This function creates interactive maps that will be displayed in RStudio’s 'Viewer' tab. The default setting of jitter will move locations randomly within an ~50 meter radius so that overlapping locations can be identified. Set jitter = 0 to see precise locations.

Usage

table_leaflet(
  locationTbl = NULL,
  maptype = c("terrain", "roadmap", "satellite", "toner"),
  extraVars = NULL,
  jitter = 5e-04,
  ...
)

Arguments

locationTbl | Tibble of known locations.
maptype | Optional name of leaflet ProviderTiles to use, e.g. terrain.
extraVars | Character vector of addition locationTbl column names to be shown in leaflet popups.
jitter | Amount to use to slightly adjust locations so that multiple monitors at the same location can be seen. Use zero or NA to see precise locations.
... | Additional arguments passed to leaflet::addCircleMarker().

Details

The maptype argument is mapped onto leaflet "ProviderTile" names. Current mappings include:

- "roadmap" => "OpenStreetMap"
- "satellite" => "Esri.WorldImagery"
- "terrain" => "Esri.WorldTopoMap"
- "toner" => "Stamen.Toner"

If a character string not listed above is provided, it will be used as the underlying map tile if available. See https://leaflet-extras.github.io/leaflet-providers/ for a list of "provider tiles" to use as the background map.

Value

A leaflet "plot" object which, if not assigned, is rendered in Rstudio’s 'Viewer' tab.
Examples

```r
## Not run:
library(MazamaLocationUtils)

# A table with all core metadata
table_leaflet(wa_monitors_500)

# A table missing some core metadata
table_leaflet(
  wa_airfire_meta,
  extraVars = c("stateCode", "countyName", "msaName")
)

# Customizing the map
table_leaflet(
  wa_airfire_meta,
  extraVars = c("stateCode", "countyName", "msaName"),
  radius = 6,
  color = "black",
  weight = 2,
  fillColor = "red",
  fillOpacity = 0.3
)

## End(Not run)
```

---

**table_leafletAdd**  
*Add to a leaflet interactive map for known locations*

---

**Description**

This function adds a layer to an interactive map displayed in RStudio's 'Viewer' tab. The default setting of jitter will move locations randomly within an ~50 meter radius so that overlapping locations can be identified. Set jitter = 0 to see precise locations.

**Usage**

```r
table_leafletAdd(
  map = NULL,
  locationTbl = NULL,
  extraVars = NULL,
  jitter = 5e-04,
  ...
)
```

**Arguments**

- `map`  
  Leaflet map.
table_load

Load a known location table

Description

Load a tibble of known locations from the preferred directory.

The known location table must be named either `<collectionName>.rda` or `<collectionName>.csv`. If both are found, only `<collectionName>.rda` will be loaded to ensure that columns will have the proper type assigned.

Usage

```r
table_load(collectionName = NULL)
```

Arguments

collectionName  Character identifier for this table.

Value

Tibble of known locations.

See Also

`setLocationDataDir`

Examples

```r
library(MazamaLocationUtils)

# Set the directory for saving location tables
setLocationDataDir(tempdir())

# Load an example table and check the dimensions
locationTbl <- get(data("wa_monitors_500"))
dim(locationTbl)
```
table_removeColumn

Remove a column of metadata in a table

Description

Remove the column matching columnName. This function can be used in pipelines.

Usage

```r
table_removeColumn(locationTbl = NULL, columnName = NULL, verbose = TRUE)
```

Arguments

- `locationTbl`: Tibble of known locations.
- `columnName`: Name of the column to be removed.
- `verbose`: Logical controlling the generation of progress messages.

Value

Updated tibble of known locations.

See Also

- `table_addColumn`
- `table_removeColumn`

Examples

```r
library(MazamaLocationUtils)

# Starting table
locationTbl <- get(data("wa_monitors_500"))
names(locationTbl)

# Add a new column
locationTbl <-
  locationTbl %>%
  table_addColumn("AQSID")
```
names(locationTbl)

# Now remove it
locationTbl <-
  locationTbl %>%
  table_removeColumn("AQSID")

names(locationTbl)

try({
  # Cannot remove "core" metadata
  locationTbl <-
  locationTbl %>%
  table_removeColumn("longitude")
}, silent = FALSE)

---

**table_removeRecord**

*Remove location records from a table*

**Description**

Incoming `locationID` values are compared against the incoming `locationTbl` and any matches are removed.

**Usage**

`table_removeRecord(locationTbl = NULL, locationID = NULL, verbose = TRUE)`

**Arguments**

- `locationTbl`: Tibble of known locations.
- `locationID`: Vector of `locationID` strings.
- `verbose`: Logical controlling the generation of progress messages.

**Value**

Updated tibble of known locations.

**See Also**

- `table_addLocation`
- `table_addSingleLocation`
- `table_updateSingleRecord`
Examples

```
library(MazamaLocationUtils)

locationTbl <- get(data("wa_monitors_500"))
dim(locationTbl)

# Wenatchee
lon <- -120.325278
lat <- 47.423333

# Get the locationID first
locationID <- table_getLocationID(locationTbl, lon, lat, distanceThreshold = 500)

# Remove it
locationTbl <- table_removeRecord(locationTbl, locationID)
dim(locationTbl)

# Test
table_getLocationID(locationTbl, lon, lat, distanceThreshold = 500)
```

---

**table_save**  
*Save a known location table*

Description

Save a tibble of known locations to the preferred directory. If `outputType` is a vector, the known locations table will be saved to the preferred directory in multiple formats.

Usage

```
table_save(
  locationTbl = NULL,
  collectionName = NULL,
  backup = TRUE,
  outputType = "rda"
)
```

Arguments

- `locationTbl`: Tibble of known locations.
- `collectionName`: Character identifier for this table.
- `backup`: Logical specifying whether to save a backup version of any existing tables sharing `collectionName`.
- `outputType`: Vector of output formats. (Currently only "rda" or "csv" are supported.)
Details

Backup files are saved with "YYYY-mm-ddTHH:MM:SS"

Value

File path of saved file.

Examples

```r
library(MazamaLocationUtils)

# Set the directory for saving location tables
setLocationDataDir(tempdir())

# Load an example table and check the dimensions
locationTbl <- get(data("wa_monitors_500"))
dim(locationTbl)

# Save it as "table_save_example"
table_save(locationTbl, "table_save_example")

# Add a column and save again
locationTbl %>%
  table_addColumn("my_column") %>%
  table_save("table_save_example")

# Check the locationDataDir
list.files(getLocationDataDir(), pattern = "table_save_example")
```

### table_updateColumn

**Update a column of metadata in a table**

**Description**

Updates records in a location table. Records are identified by `locationID` and the data found in `locationData` is used to replace any existing value in the `columnName` column. `locationID` and `locationData` must be of the same length. Any `NA` values in `locationID` will be ignored. If `columnName` is not a named column within `locationTbl`, a new column will be created.

**Usage**

```r
table_updateColumn(
  locationTbl = NULL,
  columnName = NULL,
  locationID = NULL,
  locationData = NULL,
  verbose = TRUE
)
```
Arguments

- `locationTbl`: Tibble of known locations.
- `columnName`: Name of an existing/new column in `locationTbl` whose data will be updated/created.
- `locationID`: Vector of `locationID` strings.
- `locationData`: Vector of data to be inserted at records identified by `locationID`.
- `verbose`: Logical controlling the generation of progress messages.

Value

Updated tibble of known locations.

See Also

- `table_addColumn`
- `table_removeColumn`

Examples

```r
library(MazamaLocationUtils)

locationTbl <- get(data("wa_monitors_500"))
wa <- get(data("wa_airfire_meta"))

# We will merge some metadata from wa into locationTbl

# Record indices for wa
wa_indices <- seq(5,65,5)
wSub <- wa[wa_indices,]

locationID <-
  table_getLocationID(
    locationTbl, 
    wa_sub$longitude, 
    wa_sub$latitude, 
    distanceThreshold = 500
  )

locationData <- wa_sub$AQSID

locationTbl <-
  table_updateColumn(locationTbl, "AQSID", locationID, locationData)

# Look at the data we attempted to merge
wa$AQSID[wa_indices]

# And two columns from the updated locationTbl
locationTbl_indices <- table_getRecordIndex(locationTbl, locationID)
locationTbl[locationTbl_indices, c("city", "AQSID")]
```
**table_updateSingleRecord**

*Update a single known location record in a table*

**Description**

Information in the `locationList` is used to replace existing information found in `locationTbl`. This function can be used for small tweaks to an existing `locationTbl`. Wholesale replacement of records should be performed with `table_removeRecord()` followed by `table_addLocation()`.

**Usage**

```r
library(MazamaLocationUtils)
locationTbl <- get(data("wa_monitors_500"))
# Wenatchee
wenatcheeRecord <- locationTbl %>%
  dplyr::filter(city == "Wenatchee")
str(wenatcheeRecord)
```
validateMazamaSpatialUtils

validateLocationTbl

**Validate a location table**

**Description**

Ensures that the incoming table has numeric longitude and latitude columns.

**Usage**

```r
validateLocationTbl(locationTbl = NULL, locationOnly = TRUE)
```

**Arguments**

- `locationTbl` Tibble of known locations.
- `locationOnly` Logical specifying whether to check for all standard columns.

**Value**

Invisibly returns TRUE if no error message has been generated.

validateMazamaSpatialUtils

**Validate proper setup of MazamaSpatialUtils**

**Description**

The `MazamaSpatialUtils` package must be properly installed and initialized before using functions from the `MazamaLocationUtils` package. This function tests for this.
Usage

validateMazamaSpatialUtils()

Value

Invisibly returns TRUE if no error message has been generated.

Description

The `wa_pwfsl_meta` dataset provides a set of Washington state air quality monitor metadata used by the USFS AirFire group. This dataset was generated on 2023-10-24 by running:

```r
library(AirMonitor)

wa_airfire_meta <-
    airnow_loadLatest() %>%
    monitor_filter(stateCode == "WA") %>%
    monitor_getMeta() %>%
    # On 2023-10-24, this metadata still uses zip instead of postalCode
dplyr::rename(postalCode = zip) %>%
    # Remove internal fields
dplyr::select(-dplyr::starts_with("airnow_"))

save(wa_airfire_meta, file = "data/wa_airfire_meta.rda")
```

Usage

`wa_airfire_meta`

Format

A tibble with 92 rows and 29 columns of data.
wa_monitors_500 Washington monitor locations dataset

Description
The wa_monitor_500 dataset provides a set of known locations associated with Washington state air quality monitors. This dataset was generated on 2023-10-24 by running:

```r
library(AirMonitor)
library(MazamaLocationUtils)
initializeMazamaSpatialUtils()
setLocationDataDir("./data")

monitor <- monitor_loadLatest() %>% monitor_filter(stateCode == "WA")
lons <- monitor$meta$longitude
lats <- monitor$meta$latitude

table_initialize() %>%
  table_addLocation(
    lons, lats,
    distanceThreshold = 500,
    elevationService = "usgs",
    addressService = "photon"
  ) %>%
table_save("wa_monitors_500")
```

Usage
wa_monitors_500

Format
A tibble with 78 rows and 13 columns of data.

See Also
id_monitors_500
or_monitors_500
Index

* datasets
  - coreMetadataNames, 5
  - id_monitors_500, 6
  - or_monitors_500, 18
  - wa_airfire_meta, 51
  - wa_monitors_500, 52

* environment
  - APIKeys, 3
  - getLocationDataDir, 6
  - LocationDataDir, 8
  - setLocationDataDir, 19

APIKeys, 3

clusterByDistance, 3, 20, 21
coreMetadataNames, 5
createLocationID, 8

godist, 31, 32, 34, 36
getAPIKey(APIKeys), 3
getLocationDataDir, 6, 8, 19

id_monitors_500, 6, 19, 52
initializeMazamaSpatialUtils, 7

location_createID, 8, 16, 23, 40
location_getCensusBlock, 9
location_getOpenCageInfo, 10
location_getSingleAddress_Photon, 12
location_getSingleAddress_TexasAM, 13
location_getSingleElevation_USGS, 14
location_initialize, 15
LocationDataDir, 6, 8, 19

MazamaLocationUtils, 17
MazamaLocationUtils-package (MazamaLocationUtils), 17

or_monitors_500, 7, 18, 52

setAPIKey(APIKeys), 3

setLocationDataDir, 6, 8, 19, 43
showAPIKeys(APIKeys), 3

table_addClustering, 4, 20
table_addColumn, 21, 44, 48
table_addCoreMetadata, 22
table_addLocation, 24, 28, 45, 49
table_addOpenCageInfo, 25
table_addSingleLocation, 25, 27, 45, 49
table_filterByDistance, 29
table_findAdjacentDistances, 30
table_findAdjacentLocations, 31
table_getDistanceFromTarget, 32
table_getLocationID, 33

table_getNearestDistance, 35

table_getNearestLocation, 36

table_getRecordIndex, 37

table_initialize, 38

table_initializeExisting, 39

table_leaflet, 41
table_leafletAdd, 42

table_load, 43

table_removeColumn, 22, 44, 44, 48

table_removeColumn, 4, 25, 28, 45, 49

table_save, 46

table_updateColumn, 22, 47

table_updateSingleRecord, 25, 28, 45, 49

validateLocationTbl, 50
validateMazamaSpatialUtils, 50

wa_airfire_meta, 51
wa_monitors_500, 7, 19, 52