### R topics documented:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>coreMetadataNames</td>
<td>3</td>
</tr>
<tr>
<td>getAPIKey</td>
<td>3</td>
</tr>
<tr>
<td>getLocationDataDir</td>
<td>3</td>
</tr>
<tr>
<td>id_monitors_500</td>
<td>4</td>
</tr>
<tr>
<td>LocationDataDir</td>
<td>5</td>
</tr>
<tr>
<td>location_createID</td>
<td>5</td>
</tr>
<tr>
<td>location_getCensusBlock</td>
<td>6</td>
</tr>
<tr>
<td>location_getSingleAddress_Photon</td>
<td>7</td>
</tr>
<tr>
<td>location_getSingleAddress_TexasAM</td>
<td>9</td>
</tr>
<tr>
<td>location_getSingleElevation_USGS</td>
<td>10</td>
</tr>
<tr>
<td>location_initialize</td>
<td>11</td>
</tr>
<tr>
<td>MazamaLocationUtils</td>
<td>12</td>
</tr>
<tr>
<td>mazama_initialize</td>
<td>14</td>
</tr>
<tr>
<td>or_monitors_500</td>
<td>15</td>
</tr>
<tr>
<td>setAPIKey</td>
<td>15</td>
</tr>
<tr>
<td>setLocationDataDir</td>
<td>16</td>
</tr>
<tr>
<td>showAPIKeys</td>
<td>16</td>
</tr>
<tr>
<td>table_addColumn</td>
<td>17</td>
</tr>
<tr>
<td>table_addCoreMetadata</td>
<td>18</td>
</tr>
<tr>
<td>table_addLocation</td>
<td>19</td>
</tr>
<tr>
<td>table_addSingleLocation</td>
<td>20</td>
</tr>
<tr>
<td>table_findAdjacentDistances</td>
<td>22</td>
</tr>
<tr>
<td>table_findAdjacentLocations</td>
<td>23</td>
</tr>
<tr>
<td>table_getLocationID</td>
<td>24</td>
</tr>
<tr>
<td>table_getNearestDistance</td>
<td>25</td>
</tr>
<tr>
<td>table_getNearestLocation</td>
<td>27</td>
</tr>
<tr>
<td>table_getRecordIndex</td>
<td>28</td>
</tr>
<tr>
<td>table_initialize</td>
<td>29</td>
</tr>
<tr>
<td>table_initializeExisting</td>
<td>30</td>
</tr>
<tr>
<td>table_leaflet</td>
<td>31</td>
</tr>
<tr>
<td>table_leafletAdd</td>
<td>33</td>
</tr>
<tr>
<td>table_load</td>
<td>34</td>
</tr>
<tr>
<td>table_removeColumn</td>
<td>35</td>
</tr>
<tr>
<td>table_removeRecord</td>
<td>36</td>
</tr>
<tr>
<td>table_save</td>
<td>37</td>
</tr>
<tr>
<td>table_updateColumn</td>
<td>38</td>
</tr>
<tr>
<td>table_updateSingleRecord</td>
<td>39</td>
</tr>
<tr>
<td>validateLocationTbl</td>
<td>41</td>
</tr>
<tr>
<td>validateMazamaSpatialUtils</td>
<td>41</td>
</tr>
<tr>
<td>wa_airfire_meta</td>
<td>42</td>
</tr>
<tr>
<td>wa_monitors_500</td>
<td>42</td>
</tr>
</tbody>
</table>

**Index** 44
**coreMetadataNames**

*Names of standard spatial metadata columns*

**Description**

Character string identifiers of the different types of spatial metadata this package can generate.

**Usage**

```r
coreMetadataNames
```

**Format**

A vector with 3 elements

**Details**

```r
coreMetadataNames
```

---

**getAPIKey**

*Get API key*

**Description**

Returns the API key associated with a web service.

**getLocationDataDir**

*Get location data directory*

**Description**

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

**Usage**

```r
getLocationDataDir()
```

**Value**

Absolute path string.

**See Also**

*LocationDataDir*

*setLocationDataDir*
**id_monitors_500**

Idaho monitor locations dataset

### Description

The `id_monitor_500` dataset provides a set of known locations associated with Idaho state air quality monitors. This dataset was generated on 2021-10-19 by running:

```r
library(PWFSLSmoke)
library(MazamaLocationUtils)
mazama_initialize()
setLocationDataDir("./data")

monitor <- monitor_loadLatest()
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")
```

### Usage

`id_monitors_500`

### Format

A tibble with 30 rows and 13 columns of data.

### See Also

- `or_monitors_500`
- `wa_monitors_500`
**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. The following code is used to generate each locationID. See the references for details.

```r
# Retain accuracy up to ~.1m
locationString <- paste0(
  sprintf("%.7f", longitude),
  "_",
  sprintf("%.7f", latitude)
)

# Avoid collisions until billions of records
locationID <- digest::digest(locationString, algo = "xxhash64")
```

**Usage**

`location_createID(longitude = NULL, latitude = NULL)`

**Arguments**

- **longitude**: Vector of longitudes in decimal degrees E.
- **latitude**: Vector of latitudes in decimal degrees N.
Value

Vector of character locationIDs.

References

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

Examples

```r
library(MazamaLocationUtils)

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

---

**location_getCensusBlock**

*Get census block data from the FCC API*

**Description**

The FCC Block API is used to 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.

**Usage**

```r
location_getCensusBlock(longitude = NULL, latitude = NULL, verbose = TRUE)
```

**Arguments**

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

**Value**

List of census block/county/state data.
location_getSingleAddress_Photon

References

https://geo.fcc.gov/api/census/#!/block/get_block_find

Examples

library(MazamaLocationUtils)

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

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

location_getSingleAddress_Photon

Get address data from the Photon API to OpenStreetMap

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
- zip
- 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.
location_getSingleAddress_Photon

Usage

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.

References

https://photon.komoot.io

Examples

library(MazamaLocationUtils)

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

  # Wenatchee
  lon <- -120.325278
  lat <- 47.423333

  addressList <- location_getSingleAddress_Photon(lon, lat)
  str(addressList)
}, silent = FALSE)
Get an address from a Texas A&M web service

Description

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

Usage

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. Default: NULL
- **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({

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

  # 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 associated with the longitude and latitude.

**Usage**

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

**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://nationalmap.gov/epqs/](https://nationalmap.gov/epqs/)

**Examples**

```r
library(MazamaLocationUtils)

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

lon <- -120.325278
lat <- 47.423333

d characters
    lon, lat

}, 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
- timezone
- houseNumber
- street
- city
- zip

Usage

location_initialize(
    longitude = NULL,
    latitude = NULL,
    stateDataset = "NaturalEarthAdm1",
    elevationService = NULL,
    addressService = NULL,
    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. 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

Tibble with a single new known location.

Examples

library(MazamaLocationUtils)

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

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

  # Wenatchee
  lon <- -120.325278
  lat <- 47.423333

  locationRecord <- location_initialize(lon, lat)
  str(locationRecord)
}, silent = FALSE)

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 sensorID that uniquely identify the spatial location and specific instrument making measurements. A unique timeseriesID could be produced as locationID_sensorID. Metadata associated with each time series would contain basic information needed for downstream analysis including at least:

- timeseriesID, locationID, sensorID, longitude, latitude, ...

  - Multiple sensors placed at a location could be grouped by locationID.
  - An extended timeservers for a mobile sensor would group by sensorID.
  - 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 sensorID or an associated non-spatial identifier as a standin 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 _repositioned_ 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 timeseries.
- 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.
mazama_initialize  Initialize with MazamaScience standard directories

Description

Convenience function to initialize spatial data. Wraps the following setup lines:

MazamaSpatialUtils::setSpatialDataDir(spatialDataDir)
MazamaSpatialUtils::loadSpatialData("EEZCountries")
MazamaSpatialUtils::loadSpatialData("OSMTimezones")
MazamaSpatialUtils::loadSpatialData("NaturalEarthAdm1")
MazamaSpatialUtils::loadSpatialData("USCensusCounties")

Usage

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

Arguments

spatialDataDir  Directory where spatial datasets are found, Default: "~/Data/Spatial"

Value

No return value.

Examples

library(MazamaLocationUtils)

# Fail gracefully if any resources are not available
try{

  # Set up directory for spatial data
  spatialDataDir <- tempdir()  # typically "~/Data/Spatial"
  MazamaSpatialUtils::setSpatialDataDir(spatialDataDir)

  exists("NaturalEarthAdm1")
  mazama_initialize(spatialDataDir)
  exists("NaturalEarthAdm1")
  class(NaturalEarthAdm1)

}, silent = FALSE)
**Description**

The or_monitor_500 dataset provides a set of known locations associated with Oregon state air quality monitors. This dataset was generated on 2021-10-19 by running:

```r
library(PWFSLSmoke)
library(MazamaLocationUtils)
mazama_initialize()
setLocationDataDir("./data")

monitor <- monitor_loadLatest()
lons <- monitor$meta$longitude
lats <- monitor$meta$latitude

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

**Usage**

or_monitors_500

**Format**

A tibble with 69 rows and 13 columns of data.

**See Also**

id_monitors_500

wa_monitors_500

---

**setAPIKey**

*Set API key*

**Description**

Sets the API key associated with a web service.
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`

---

showAPIKeys  
Show API keys

**Description**

Prints a list of all currently set API keys.
Add a new column of metadata to a table

Description
A new metadata column is added to the locationTbl. For matching locationID records the associated locationData is inserted. Otherwise, the new column will be initialized with NA.

Usage
```r
table_addColumn(
  locationTbl = NULL,
  columnName = NULL,
  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
```r
library(MazamaLocationUtils)

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

# Add an empty column
locationTbl <-
  locationTbl %>%
  table_addColumn("siteName")
```
table_addCoreMetadata

Addes missing metadata columns a known location table

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
- zip

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. These columns must then be filled in to create a usable "known locations" table.

Usage

```r
table_addCoreMetadata(locationTbl = NULL)
```

Arguments

- `locationTbl` Tibble of known locations. This input tibble need not be a standardized "known location" with all required columns. They will be added.
**table_addLocation**

**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 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. 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.
**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`.

**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"
  mazama_initialize(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)
```
Usage

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

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"
mazama_initialize(spatialDataDir)

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

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

locationTbl <-
locationTbl %>%
  table_addSingleLocation(lon, lat, distanceThreshold = 500)
}, silent = FALSE)

##

### 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.

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 = "geodesic"
)

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`.

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)
```

---

table_findAdjacentLocations

Finds adjacent locations in a known locations table.

Description

Calculates distances between all locations within a known locations table and returns a tibble with the row indices and separation distances of those records separated by less than distanceThreshold meters.

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 = "geodesic"
)
```

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.
Value

Tibble of known 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?
meta %>%
  table_findAdjacentLocations(distanceThreshold = 2000) %>%
  dplyr::select(monitorID, siteName, timezone)

# How about 4 km?
meta %>%
  table_findAdjacentLocations(distanceThreshold = 4000) %>%
  dplyr::select(monitorID, siteName, timezone)
```

---

### 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
table_getLocationID(
  locationTbl = NULL,
  longitude = NULL,
  latitude = NULL,
  distanceThreshold = NULL,
  measure = "geodesic"
)
```
table_getNearestDistance

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

Description

Returns distances from known locations in `locationTbl`, one for each incoming location. If no known location is found within `distanceThreshold` meters for a particular incoming location, that distance in the vector will be NA.
Usage

table_getNearestDistance(
    locationTbl = NULL,
    longitude = NULL,
    latitude = NULL,
    distanceThreshold = NULL,
    measure = "geodesic"
)

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

Value

Vector of distances from known locations.

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

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 known locations from `locationTbl`, one for each incoming 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.
- `latitude` Vector of latitudes in decimal degrees N.
- `distanceThreshold` Distance in meters.

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()
```
table_getRecordIndex

Return indexes of known location records

Description

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

Usage

```r
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
- zip

Usage

table_initialize()

Value

Empty known location tibble with the specified metadata columns.

Examples

library(MazamaLocationUtils)

# Create an empty Tbl
emptyTbl <- table_initialize()
dplyr::glimpse(emptyTbl)
table_initializeExisting

Converts an existing table into a known location table

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
- zip

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 with that data. Any additional columns of information 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 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

```r
table_initializeExisting(
  locationTbl = NULL,
  stateDataset = "NaturalEarthAdm1",
  countryCodes = NULL,
  distanceThreshold = NULL,
  measure = "geodesic",
  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.

- **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 these issue 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**

```r
table_leaflet(
  locationTbl = NULL,  
  maptype = "terrain",  
  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:

1. "roadmap" – "OpenStreetMap"
2. "satellite" – "Esri.WorldImagery"
3. "terrain" – "Esri.WorldTopoMap"
4. "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/](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
)```
Description

This function adds 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_leafletAdd(
  map = NULL,
  locationTbl = NULL,
  extraVars = NULL,
  jitter = 5e-04,
  ...
)

Arguments

map Leaflet map.
locationTbl Tibble of known locations.
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::addCircleMarkers().

Value

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

Load a known location table

Description

Load a tibble of known locations from the preferred directory.

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)

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

# Load it and check the dimensions
my_table <- table_load("table_load_example")
dim(my_table)

# Check the locationDataDir
list.files(getLocationDataDir(), pattern = "table_load_example")
```
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("siteName")
names(locationTbl)

# Now remove it
locationTbl <-
  locationTbl %>%
  table_removeColumn("siteName")
names(locationTbl)
```
try({
  # Cannot remove "core" metadata
  locationTbl <-
  locationTbl %>%
  table_removeColumn("zip")
}, 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**

```r
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**

```r
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.

**Usage**

```r
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`: Output format, Default: 'rda'

**Details**

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

**Value**

File path of saved file.
Examples

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")

description

Update a column of metadata in a table

For matching locationID records the associated locatioData is used to replace any existing value in columnName. NA values in locationID will be ignored.

Usage

table_updateColumn(
  locationTbl = NULL,
  columnName = NULL,
  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.
table_updateSingleRecord

**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)
wa_sub <- wa[wa_indices,]

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

locationData <- wa_sub$siteName

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

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

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

---

**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
table_updateSingleRecord(
  locationTbl = NULL,
  locationList = NULL,
  verbose = TRUE
)
```

Arguments

- `locationTbl` Tibble of known locations.
- `locationList` List containing ‘locationID’ and one or more named columns whose values are to be replaced.
- `verbose` Logical controlling the generation of progress messages.

Value

Updated tibble of known locations.

See Also

- `table_addLocation`
- `table_addSingleLocation`
- `table_removeRecord`

Examples

```r
library(MazamaLocationUtils)

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

# Wenatchee
wenatcheeRecord <-
  locationTbl %>%
  dplyr::filter(city == "Wenatchee")

str(wenatcheeRecord)

wenatcheeID <- wenatcheeRecord$locationID

locationTbl <- table_updateSingleRecord(
  locationTbl,
  locationList = list(
    locationID = wenatcheeID,
```
validateLocationTbl

locationName = "Wenatchee-Fifth St"

# Look at the new record
locationTbl %>%
dplyr::filter(city == "Wenatchee") %>%
str()

---

validateLocationTbl  Validate a location table

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

Usage
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. Functions can test for this

Usage
validateMazamaSpatialUtils()

Value
Invisibly returns TRUE if no error message has been generated.
The wa_monitors_500 dataset provides a set of Washington state air quality monitor metadata used by the USFS AirFire group. This dataset was generated on 2021-10-19 by running:

```r
library(PWFSLSmoke)

wa_monitors_500 <-
    monitor_loadLatest()
    monitor_subset(stateCodes = "WA")
    monitor_extractMeta()

save(wa_monitors_500, file = "data/wa_monitors_500.rda")
```

Usage

wa_monitors_500

Format

A tibble with 500 rows and 19 columns of data.

The wa_monitors_500 dataset provides a set of known locations associated with Washington state air quality monitors. This dataset was generated on 2021-10-19 by running:

```r
library(PWFSLSmoke)
library(MazamaLocationUtils)

mazama_initialize()
setLocationDataDir("./data")

monitor <- monitor_loadLatest()
lons <- monitor$meta$longitude
lats <- monitor$meta$latitude

table_initialize()
table_addLocation(
```
wa_monitors_500

```r
lons, lats,
distanceThreshold = 500,
elevationService = "usgs",
addressService = "photon"
)
table_save("wa_monitors_500")
```

Usage

`wa_monitors_500`

Format

A tibble with 72 rows and 13 columns of data.

See Also

- `id_monitors_500`
- `or_monitors_500`
Index

* datasets
  - coreMetadataNames, 3
  - id_monitors_500, 4
  - or_monitors_500, 15
  - wa_airfire_meta, 42
  - wa_monitors_500, 42

* environment
  - getLocationDataDir, 3
  - LocationDataDir, 5
  - setLocationDataDir, 16

- coreMetadataNames, 3
- getAPIKey, 3
- getLocationDataDir, 3, 5, 16
- id_monitors_500, 4, 15, 43

- location_createID, 5
- location_getCensusBlock, 6
- location_getSingleAddress_Photon, 7
- location_getSingleAddress_TexasAM, 9
- location_getSingleElevation_USGS, 10
- location_initialize, 11
- LocationDataDir, 3, 5, 16

- mazama_initialize, 14
- MazamaLocationUtils, 12

- or_monitors_500, 4, 15, 43

- setAPIKey, 15
- setLocationDataDir, 3, 5, 16, 34
- showAPIKeys, 16

- table_addColumn, 17, 35, 39
- table_addCoreMetadata, 18
- table_addLocation, 19, 21, 36, 40
- table_addSingleLocation, 20, 20, 36, 40
- table_findAdjacentDistances, 22
- table_findAdjacentLocations, 23
- table_getLocationID, 24
- table_getNearestDistance, 25
- table_getNearestLocation, 27
- table_getRecordIndex, 28
- table_initialize, 29
- table_initializeExisting, 30
- table_leaflet, 31
- table_leafletAdd, 33
- table_load, 34
- table_removeColumn, 17, 35, 35, 39
- table_removeRecord, 20, 21, 36, 40
- table_save, 37
- table_updateColumn, 17, 38
- table_updateSingleRecord, 20, 21, 36, 39

- validateLocationTbl, 41
- validateMazamaSpatialUtils, 41

- wa_airfire_meta, 42
- wa_monitors_500, 4, 15, 42