# Package 'EJAM'

February 12, 2023

**Title** EJAM Environmental Justice Analysis Multisite tool

**Version** 2.1.1

**Author** Mark Corrales

**Maintainer** Mark Corrales <corrales.mark@epa.gov>

**License** MIT + file LICENSE.md

**Description** Tools for summarizing environmental and demographic indicators (such as those in EJSCREEN) for residents living near any one of a number of specific sites. It uses quad tree search/indexing of block locations, data.table, parallel processing to provide very fast identification of nearby blocks, distances, and aggregation of indicators within each distance. It can be uses as a web app, with the user interface provided by the shiny R package.

**URL** https://github.com/USEPA/EJAM

**Depends** R (>= 2.10),
shiny (>= 1.7.2),
data.table,
EJAMblockdata,
EJAMfrsdata

**Imports** config (>= 0.3.1),
golem (>= 0.3.3),
foreach,
sp,
SearchTrees,
pdist,
DBI,
RMySQL,
doSNOW,
EJAMbatch.summarizer,
EJAMejscreendata,
EJAMejscreenapi,
openxlsx,
pkgload

**Remotes** github::USEPA/EJAM,
github::USEPA/EJAMblockdata,
github::USEPA/EJAMfrsdata,
github::USEPA/EJAMbatch.summarizer,
github::USEPA/EJAMejscreendata,
github::USEPA/EJAMejscreenapi

**Suggests** knitr,
rmarkdown,
spelling,
testthat (>= 3.0.0)

**Config/testthat/edition** 3

**Encoding** UTF-8

**LazyData** true

**Language** en-US

**VignetteBuilder** knitr

**RoxygenNote** 7.2.3

**Roxygen** list(markdown = TRUE)

# R topics documented:

---

`.onLoad` *Creates index to all US blocks (internal point lat lon) at package load*

---

## Description

Creates index to all US blocks (internal point lat lon) at package load

## Usage

```
.onLoad(libname, pkgname)
```

## Arguments

| | |
|---|---|
| `libname` | na |
| `pkgname` | na |

---

`all.equal_functions` *helper function for checking possibly different versions of a function with same name in 2 packages*

---

## Description

helper function for checking possibly different versions of a function with same name in 2 packages

## Usage

```
## S3 method for class 'equal_functions'
all(fun = "latlon_infer", package1 = "EJAM", package2 = "EJAMejscreenapi")
```

### Arguments

| | |
|---|---|
| fun | quoted name of function, like "latlon_infer" |
| package1 | quoted name of package, like "EJAM" |
| package2 | quoted name of package, like "EJAMejscreenapi" |

### Value

TRUE or FALSE

### See Also

dupenames()

---

app_run_EJAM *Run the Shiny Application app_run_EJAM() is like EJAM::run_app() app_run_EJAMejscreenapi() is like EJAMejscreenapi::run_app() app_run_EJAMbatch.summarizer() is like EJAMbatch.summarizer::run_app()*

---

### Description

Run the Shiny Application app_run_EJAM() is like EJAM::run_app() app_run_EJAMejscreenapi() is like EJAMejscreenapi::run_app() app_run_EJAMbatch.summarizer() is like EJAMbatch.summarizer::run_app()

### Usage

```
app_run_EJAM(
  onStart = NULL,
  options = list(),
  enableBookmarking = NULL,
  uiPattern = "/",
  ...
)
```

### Arguments

| | |
|---|---|
| onStart | A function that will be called before the app is actually run. This is only needed for shinyAppObj, since in the shinyAppDir case, a global.R file can be used for this purpose. |
| options | Named options that should be passed to the runApp call (these can be any of the following: "port", "launch.browser", "host", "quiet", "display.mode" and "test.mode"). You can also specify width and height parameters which provide a hint to the embedding environment about the ideal height/width for the app. |
| enableBookmarking | Can be one of "url", "server", or "disable". The default value, NULL, will respect the setting from any previous calls to enableBookmarking(). See enableBookmarking() for more information on bookmarking your app. |

| | |
|---|---|
| uiPattern | A regular expression that will be applied to each GET request to determine whether the ui should be used to handle the request. Note that the entire request path must match the regular expression in order for the match to be considered successful. |
| ... | arguments to pass to golem_opts. See ?golem::get_golem_options for more details. |

---

bgpts *lat lon of popwtd center of blockgroup, and count of blocks per block group*

---

## Description

This is just a list of US block groups and how many blocks are in each... It also has the lat and lon roughly of each blockgroup

## Details

The point used for each bg is the Census 2020 population weighted mean of the blocks' internal points. It gives an approximation of where people live and where each bg is, which is useful for some situations.

```
  As of 10/2022 it is the EJScreen 2.1 version of data, which uses ACS 2016-2020
  and Census 2020. it has all US States, DC, PR, but not  "AS" "GU" "MP" "VI"

  How lat lon were estimated:

# proxistat::bg.pts had a lat/lon internal point for each us block group for Census 2010.
# that had been used to include those lat/lon in ejscreen::bg21, for convenience.
 > head(proxistat::bg.pts)
           FIPS    aland   awater      lat       lon
           1 010950302024 14969994 15040133 34.42668 -86.2437
            2 010950306002  6751877 16610261 34.31763 -86.34399

 # Now, for Census 2020 blocks, create pop wtd centroids lat lon for each block group ####
 #  using EJAMblockdata::blockwts and EJAMblockdata::blockpoints

 bgpts_blocks <- copy(blockpoints) # not essential but ok to make sure we do not change blockpoints it
 # all.equal(bgpts$blockid , blockwts$blockid)
 bgpts_blocks[ , bgid    := blockwts$bgid]
 bgpts_blocks[ , blockwt := blockwts$blockwt]
 # get pop wtd mean of lat, and same for lon, by bgid
 bgpts <- bgpts_blocks[ , lapply(.SD, FUN = function(x) stats::weighted.mean(x, w = blockwt, na.rm =
 rm( bgpts_blocks)
 # add the bgfips column, so it has bgfips, bgid, lat, lon
 # all.equal(bgpts$bgid,bgid2fips$bgid)
 bgpts[ , bgfips := bgid2fips$bgfips]
 # setnames(bgpts, 'bgfips', 'FIPS')

 # BUT NOTE this census2020 block table has PR but lacks "AS" "GU" "MP" "VI" ####
```

```
# > uniqueN(EJAMblockdata::blockid2fips[,substr(blockfips,1,2)])
# [1] 52
# length(unique(EJAMejscreendata::EJSCREEN_Full_with_AS_CNMI_GU_VI$ST_ABBREV))
# [1] 56
#   dim(bgejam)
# [1] 242,940    155
#   dim(bg22)
# [1] 242,335    157
#
# so how do we get latlon for bg in as/gu/mp/vi ?  ?####

# view those block group points on a map (plot only a subset which is enough)
sam <- sample(seq_along(bgpts$bgid),5000)
plot(bgpts$lon[sam], bgpts$lat[sam], pch = '.')

# view one state, florida, where 12 are the 1st 2 digits of the FIPS:
# bgpts[bgid2fips[substr(bgfips,1,2) == '12', ], on = 'bgid']
xx='12'
mystate <- bgpts[bgid2fips[substr(bgfips, 1, 2) == xx, ], on = 'bgid'][ , .(lon,lat)]
plot(mystate, pch = '.')
rm(mystate, xx)


 How blockcounts were done:

 library(EJAMblockdata)
 library(data.table)
 bg_blockcounts <- blockwts[ , .(blockcount = uniqueN(.SD)), by=bgid]
 sum(bg_blockcounts$blockcount == 1)
   # [1] 1874 blockgroups have only 1 block
 sum(bg_blockcounts$blockcount == 1000)  the max is 1000 blocks in a bg
   # # [1] 22
round(100*table(bg_blockcounts[blockcount <20, blockcount]) / nrow(bg_blockcounts) ,1)
   # about 1 to 3
   #   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
   # 0.8 1.2 1.3 1.4 1.5 2.1 2.2 2.4 2.6 2.8 2.8 3.0 3.0 2.9 3.0 2.9 2.8 2.7 2.5
   all.equal(bgpts$bgid, bg_blockcounts$bgid)
 bgpts[ , blockcount := bg_blockcounts$blockcount]
 dim(bgpts)
     # 242335  x    5
 usethis::use_data(bgpts) # saved for EJAM package
```

---

`blockgroupstats` *EJScreen demographic and enviromental indicators for Census block groups*

---

## Description

The EJScreen dataset (demographic, environmental, EJ indicators), plus more demographic subgroups.

### Details

As of 10/2022 it is the EJScreen 2.1 version of data, which uses ACS 2016-2020. EJScreen 2.1 was released October 2022. As of 4/2022 it was the EJScreen 2.0 version of data, which used ACS 2015-2019. EJScreen 2.0 was released 2/18/2022 (raw data download avail 2/22/2022).

NOTE: It also has the race/ethnic subgroups that add up to minority or people of color.

Each year this could be created as for the latest version. See attributes(blockgroupstats) It is also available in a similar form via the ejscreen package on github, and EJAMejscreendata::EJSCREEN_Full_with_AS_CNM but there are differences in which columns are kept.

It is a data.table of US Census blockgroups (not blocks). With PR, 242,335 rows, approx 175 columns. See https://www.epa.gov/ejscreen

column names include bgfips, bgid (for join to blockwt$bgid), pop, pctlowinc, pcthisp, etc.

see source code and notes in EJAM/inst/notes_datasets/ which has create_blockgroupstats()

See maybe the notes on cleaning up and changing the dataset starting from ejscreen::bg22plus

---

## computeActualDistancefromSurfacedistance

*convert surface distance to actual distance*

---

### Description

\preformatted{
Just a simple formula:
earthRadius_miles <- 3959
angle_rad <- x/earthRadius_miles
# Calculate radius * cord length
return( earthRadius_miles * 2*sin(angle_rad/2) )
}

### Usage

```
computeActualDistancefromSurfacedistance(x)
```

### Arguments

| | |
|---|---|
| x | surface distance in miles |

---

## datapack

*See info about the data sets in one or more packages Wrapper for data() - just shows info in console and silently returns a data.frame*

---

### Description

See info about the data sets in one or more packages Wrapper for data() - just shows info in console and silently returns a data.frame

### Usage

```
datapack(pkg = ejampackages, len = 30)
```

### Arguments

| | |
|---|---|
| pkg | a character vector giving the package(s) to look in for data sets |
| len | Only affects what is printed to console - specifies the number of characters to limit Title to, making it easier to see in the console. |

### Value

data.frame with Item and Title as columns

### Examples

```
datapack("datasets")
datapack("MASS")
datapack(ejampackages)
```

---

## doaggregate *Summarize indicators in each buffer (given the blocks in each buffer and indicators for each block)*

---

### Description

This updated 2022 code takes a set of facilities and the set of blocks that are near each, (as identified previously, in other code that has identified which blocks are nearby) and combines those with indicator scores for block groups.

### Usage

```
doaggregate(
  sites2blocks,
  countcols = NULL,
  popmeancols = NULL,
  calculatedcols = NULL,
  testing = FALSE,
  ...
)
```

### Arguments

| | |
|---|---|
| sites2blocks | data.table of distances in miles between all sites (facilities) and nearby Census block internal points, with columns siteid, blockid, distance, created by getblocksnearby function. See sites2blocks_example dataset in package, as input to this function |
| countcols | character vector of names of variables to aggregate within a buffer using a sum of counts, like, for example, the number of people for whom a poverty ratio is known, the count of which is the exact denominator needed to correctly calculate percent low income. |
| popmeancols | character vector of names of variables to aggregate within a buffer using population weighted mean. |
| calculatedcols | character vector of names of variables to aggregate within a buffer using formulas that have to be specified. |
| testing | used while testing this function |
| ... | more to pass to another function? Not used currently. |

### Details

This function aggregates the blockgroup scores to create a summary of each indicator, as a raw score and US percentile and State percentile, in each buffer (i.e., near each facility):

- **SUMS OF COUNTS**: for population count, or number of households or Hispanics, etc.
- **POPULATION-WEIGHTED MEANS**: for Environmental indicators.
  **EJ Indexes***:* These could be in theory recalculated via formula, but the way EJScreen does this is apparently finding the pop wtd mean of EJ Index raw scores, not the EJ Index formula applied to the summarized demographic score and aggregated envt number.
- **CALCULATED BY FORMULA**: Buffer or overall score calculated via formulas using aggregated counts, such as percent low income = sum of counts low income / sum of counts of denominator, which in this case is the count of those for whom the poverty ratio is known.
- **LOOKED UP**: Aggregated scores are converted into percentile terms via lookup tables (US or State version).

This function requires the following as data lazy loaded for example from EJAMblockdata package:

- blockwts: data.table with these columns: blockid , bgid, blockwt
- quaddata, and blockquadtree: data.table and quad tree, for indexes of block points (and local-tree that is created when package is loaded)
- EJAM::blockgroupstats - A data.table (such as EJSCREEN demographic and environmental data by blockgroup?)

---

`dupenames` *helper function to look at several packages to spot conflicting exported names See what objects (functions or data) are exported by a given (installed) package*

---

### Description

helper function to look at several packages to spot conflicting exported names See what objects (functions or data) are exported by a given (installed) package

### Usage

```
dupenames(
  pkg = EJAM::ejampackages,
  sortbypkg = FALSE,
  compare.functions = TRUE
)
```

### Arguments

| | |
|---|---|
| `pkg` | one or more package names as vector of strings. If "all" it checks all installed pkgs, but takes very very long potentially. |
| `sortbypkg` | If TRUE, just returns same thing but sorted by package name |
| `compare.functions` | If TRUE, sends to console inf about whether body and formals of the functions are identical between functions of same name from different packages. Only checks the first 2 copies, not any additional ones (where 3+ pkgs use same name) |

## Details

This can help find duplicates/conflicts within source code and make sure they are on search path, for when renaming / moving functions/packages

## Value

data.frame with columns Package, Object name (or NA if no dupes)

## See Also

`all.equal_functions()`

---

# ejampackages *list of names of EJAM-related R packages*

---

## Description

list of names of EJAM-related R packages

---

# getacs_epaquery *experimental/ work in progress: get ACS data via EPA API (for <200 places)*

---

## Description

uses ACS2019 rest services ejscreen ejquery MapServer 7

Documentation of format and examples of input parameters: https://geopub.epa.gov/arcgis/sdk/rest/index.html#/Query_M

## Usage

```
getacs_epaquery(
  objectIds = 1:3,
  servicenumber = 7,
  outFields = NULL,
  returnGeometry = FALSE,
  justurl = FALSE,
  ...
)
```

## Arguments

| | |
|---|---|
| `objectIds` | see API |
| `servicenumber` | see API |
| `outFields` | see API. eg "STCNTRBG","TOTALPOP","PCT_HISP", |
| `returnGeometry` | see API |
| `justurl` | if TRUE, returns url instead of default making API request |
| `...` | passed to getacs_epaquery_chunked() |

## Value

table

## Examples

```
getacs_epaquery(justurl=TRUE)
```

---

`getacs_epaquery_chunked` *experimental/ work in progress: in chunks, get ACS data via EPA API*

---

## Description

experimental/ work in progress: in chunks, get ACS data via EPA API

## Usage

```
getacs_epaquery_chunked(
  objectIds = 1:3,
  servicenumber = 7,
  outFields = NULL,
  returnGeometry = FALSE,
  justurl = FALSE,
  chunksize = 200,
  ...
)
```

## Arguments

| | |
|---|---|
| `objectIds` | see API |
| `servicenumber` | see API |
| `outFields` | see API |
| `returnGeometry` | see API |
| `justurl` | see API |
| `chunksize` | eg 200 for chunks of 200 each request |
| `...` | passed to getacs_epaquery() |

## Value

table

## Examples

```
#
#\dontrun {
# x <- list() # chunked chunks. best not to ask for all these:
# x[[1]] <- getacs_epaquery_chunked(   1:1000, chunksize = 100)
# x[[2]] <- getacs_epaquery_chunked(1001:5000, chunksize = 100)
# xall <- do.call(rbind, x)
#}
```

`getblocksnearby` *Key buffering function - wrapper redirecting to the right version of getblocksnearby()*

## Description

Key buffering function - wrapper redirecting to the right version of getblocksnearby()

## Usage

```
getblocksnearby(
  sitepoints,
  cutoff = 1,
  maxcutoff = 31.07,
  avoidorphans = TRUE,
  quadtree,
  ...
)
```

## Arguments

| | |
|---|---|
| `sitepoints` | see `getblocksnearbyviaQuadTree()` or other such functions |
| `cutoff` | see `getblocksnearbyviaQuadTree()` or other such functions |
| `maxcutoff` | see `getblocksnearbyviaQuadTree()` or other such functions |
| `avoidorphans` | see `getblocksnearbyviaQuadTree()` or other such functions |
| `quadtree` | a large quadtree object created from the SearchTree package example: SearchTrees::createTree(EJAM treeType = "quad", dataType = "point") |
| `...` | see `getblocksnearbyviaQuadTree_Clustered()` or other such functions |

## See Also

`getblocksnearby2()` that was work in progress

`getblocksnearby2` *Key buffering function - wrapper redirecting to the right version of getblocksnearby()*

## Description

Key buffering function - wrapper redirecting to the right version of getblocksnearby()

## Usage

```
getblocksnearby2(
  sitepoints,
  cutoff = 1,
  maxcutoff = 31.07,
  avoidorphans = TRUE,
  quadtree = is.null,
  ...
)
```

### Arguments

| | |
|---|---|
| sitepoints | see getblocksnearbyviaQuadTree() or other such functions |
| cutoff | see getblocksnearbyviaQuadTree() or other such functions |
| maxcutoff | see getblocksnearbyviaQuadTree() or other such functions |
| avoidorphans | see getblocksnearbyviaQuadTree() or other such functions |
| quadtree | a large quadtree object created from the SearchTree package example: SearchTrees::createTree(EJAM treeType = "quad", dataType = "point") |
| ... | see getblocksnearbyviaQuadTree_Clustered() or other such functions |

### Details

Like getblocksnearby() but tries to handle localtree and quadtree parameter differently

- not sure how to check if they are in the right environment.

### See Also

getblocksnearby()

---

## getblocksnearbyviaQuadTree

*Find nearby blocks using Quad Tree data structure for speed, NO PARALLEL PROCESSING*

---

### Description

Given a set of points and a specified radius (cutoff), this function quickly finds all the US Census blocks near each point. For each point, it uses the specified cutoff distance and finds the distance to every block within the circle defined by the radius (cutoff). Each block is defined by its Census-provided internal point, by latitude and longitude.

Each point can be the location of a regulated facility or other type of site, and the blocks are a high-resolution source of information about where residents live.

Finding which blocks have their internal points in a circle provides a way to quickly estimate what fraction of a block group is inside the circular buffer more accurately and more quickly than areal apportionment of block groups would provide.

### Usage

```
getblocksnearbyviaQuadTree(
  sitepoints,
  cutoff = 1,
  maxcutoff = 31.07,
  avoidorphans = TRUE,
  report_progress_every_n = 500,
  quadtree
)
```

### Arguments

| | |
|---|---|
| sitepoints | data.table with columns siteid, lat, lon giving point locations of sites or facilities around which are circular buffers |
| cutoff | miles radius, defining circular buffer around site point |
| maxcutoff | miles distance (max distance to check if not even 1 block point is within cutoff) |
| avoidorphans | logical Whether to avoid case where no block points are within cutoff, so if TRUE, it keeps looking past cutoff to find nearest one within maxcutoff. |
| report_progress_every_n | Reports progress to console after every n points, mostly for testing, but a progress bar feature might be useful unless this is super fast. |
| quadtree | (a pointer to the large quadtree object) created from the SearchTree package example: SearchTrees::createTree(EJAMblockdata::quaddata, treeType = "quad", dataType = "point") Takes about 2-5 seconds to create this each time it is needed. It can be automatically created when the package is loaded via the .onLoad() function |

### See Also

getblocksnearbyviaQuadTree_Clustered() getblocksnearbyviaQuadTree2()

### Examples

```
localtree_example = SearchTrees::createTree(EJAMblockdata::quaddata, treeType = "quad", dataType = "point")
 x = getblocksnearby(testpoints_1000_dt, quadtree = localtree_example)
```

---

## getblocksnearbyviaQuadTree2

*Find nearby blocks using Quad Tree data structure for speed, NO PARALLEL PROCESSING*

---

### Description

This should be almost identical to getblocksnearbyviaQuadTree(), but it uses f2, a copy of sitepoints, and more importantly it pulls some code out of the for loop and uses a vectorized approach. Given a set of points and a specified radius (cutoff), this function quickly finds all the US Census blocks near each point. For each point, it uses the specified cutoff distance and finds the distance to every block within the circle defined by the radius (cutoff). Each block is defined by its Census-provided internal point, by latitude and longitude.

Each point can be the location of a regulated facility or other type of site, and the blocks are a high-resolution source of information about where residents live.

Finding which blocks have their internal points in a circle provides a way to quickly estimate what fraction of a block group is inside the circular buffer more accurately and more quickly than areal apportionment of block groups would provide.

## Usage

```
getblocksnearbyviaQuadTree2(
  sitepoints,
  cutoff = 1,
  maxcutoff = 31.07,
  avoidorphans = TRUE,
  report_progress_every_n = 500,
  quadtree
)
```

## Arguments

| | |
|---|---|
| sitepoints | data.table with columns siteid, lat, lon giving point locations of sites or facilities around which are circular buffers |
| cutoff | miles radius, defining circular buffer around site point |
| maxcutoff | miles distance (max distance to check if not even 1 block point is within cutoff) |
| avoidorphans | logical Whether to avoid case where no block points are within cutoff, so if TRUE, it keeps looking past cutoff to find nearest one within maxcutoff. |
| report_progress_every_n | Reports progress to console after every n points, mostly for testing, but a progress bar feature might be useful unless this is super fast. |
| quadtree | (a pointer to the large quadtree object) created from the SearchTree package example: SearchTrees::createTree(EJAMblockdata::quaddata, treeType = "quad", dataType = "point") Takes about 2-5 seconds to create this each time it is needed. It can be automatically created when the package is loaded via the .onLoad() function |

## See Also

getblocksnearbyviaQuadTree_Clustered() getblocksnearbyviaQuadTree()

## Examples

```
localtree_example = SearchTrees::createTree(EJAMblockdata::quaddata, treeType = "quad", dataType = "point")
 x = getblocksnearby2(testpoints_1000_dt, quadtree = localtree_example)
```

---

# getblocksnearbyviaQuadTree_Clustered

*find nearby blocks using Quad Tree data structure for speed, CLUSTERED FOR PARALLEL PROCESSING*

---

## Description

Uses packages parallel and snow. parallel::makePSOCKcluster is an enhanced version of snow::makeSOCKcluster in package snow. It runs Rscript on the specified host(s) to set up a worker process which listens on a socket for expressions to evaluate, and returns the results (as serialized objects).

## Usage

```
getblocksnearbyviaQuadTree_Clustered(
  facilities,
  cutoff,
  maxcutoff,
  avoidorphans,
  CountCPU = 1
)
```

## Arguments

| | |
|---|---|
| facilities | data.table with columns LAT, LONG |
| cutoff | miles distance (check what this actually does) |
| maxcutoff | miles distance (check what this actually does) |
| avoidorphans | logical |
| CountCPU | for parallel processing via makeCluster() and doSNOW::registerDoSNOW() |

## Details

Uses indexgridsize and quaddata variables that come from global environment (but should pass to this function rather than assume in global env?)

## See Also

getblocksnearbyviaQuadTree() computeActualDistancefromSurfacedistance()

---

# getblocksnearby_and_doaggregate

*Wrapper for getblocksnearby() plus doaggregate()*

---

## Description

Wrapper for getblocksnearby() plus doaggregate()

## Usage

```
getblocksnearby_and_doaggregate(
  sitepoints,
  cutoff = 1,
  maxcutoff = 31.07,
  avoidorphans = TRUE,
  quadtree,
  ...
)
```

### Arguments

| | |
|---|---|
| sitepoints | see getblocksnearbyviaQuadTree() or other such functions |
| cutoff | see getblocksnearbyviaQuadTree() or other such functions |
| maxcutoff | see getblocksnearbyviaQuadTree() or other such functions |
| avoidorphans | see getblocksnearbyviaQuadTree() or other such functions |
| quadtree | a large quadtree object created from the SearchTree package example: SearchTrees::createTree(EJAM treeType = "quad", dataType = "point") |
| ... | see getblocksnearbyviaQuadTree_Clustered() or other such functions |

---

## get_any_rest_chunked_by_id

*experimental/ work in progress: in chunks, get ACS data or Block weights nearby via EPA API*

---

### Description

experimental/ work in progress: in chunks, get ACS data or Block weights nearby via EPA API

### Usage

```
get_any_rest_chunked_by_id(objectIds, chunksize = 200, ...)
```

### Arguments

| | |
|---|---|
| objectIds | see API |
| chunksize | see API |
| ... | passed to getacs_epaquery() |

### Value

a table

---

## get_via_url

*helper function work in progress: GET json via url of ejscreen ejquery map services*

---

### Description

helper function work in progress: GET json via url of ejscreen ejquery map services

### Usage

```
get_via_url(url)
```

### Arguments

| | |
|---|---|
| url | the url for an EJScreen ejquery request |

### Value

json

## hasfield *helper function*

### Description

helper function

### Usage

```
hasfield(data, fieldname)
```

### Arguments

| | |
|---|---|
| data | data.table |
| fieldname | colname to check |

### Value

logical

## latlon_as.numeric *Strip non-numeric characters from a vector*

### Description

Remove all characters other than minus signs, decimal points, and numeric digits

### Usage

```
latlon_as.numeric(x)
```

### Arguments

| | |
|---|---|
| x | vector of something that is supposed to be numbers like latitude or longitude and may be a character vector because there were some other characters like tab or space or percent sign or dollar sign |

### Details

Useful if latitude or longitude vector has spaces, tabs, etc. CAUTION - Assumes stripping those out and making it numeric will fix whatever problem there was and end result is a valid set of numbers. Inf etc. are turned into NA values. Empty zero length string is turned into NA without warning. NA is left as NA. If anything other than empty or NA could not be interpreted as a number, it returns NA for those and offers a warning.

### Value

numeric vector same length as x

## See Also

latlon_df_clean() latlon_infer() latlon_is.valid() latlon_as.numeric()

## Examples

```
latlon_as.numeric(c("-97.179167000000007", " -94.0533", "-95.152083000000005"))
latlon_as.numeric(-3:3)
latlon_as.numeric(c(1:3, NA))
latlon_as.numeric(c(1, 'asdf'))
latlon_as.numeric(c(1, ''))
latlon_as.numeric(c(1, '', NA))
latlon_as.numeric(c('aword', '$b'))
latlon_as.numeric(c('-10.5%', '<5', '$100'))
latlon_as.numeric(c(Inf, 1))
```

---

# latlon_df_clean *Find and clean up latitude and longitude columns in a data.frame*

---

## Description

Utility to identify lat and lon columns, renaming and cleaning them up.

## Usage

```
latlon_df_clean(df)
```

## Arguments

| | |
|---|---|
| df | data.frame With columns lat and lon or names that can be interpreted as such - see latlon_infer() |

## Details

Tries to figure out which columns seem to have lat lon values, renames those in the data.frame. Cleans up lat and lon values (removes extra characters, makes numeric)

## Value

Returns the same data.frame but with relevant colnames changed to lat and lon, and invalid lat or lon values cleaned up if possible or else replaced with NA

## See Also

latlon_df_clean() latlon_infer() latlon_is.valid() latlon_as.numeric()

## Examples

```
#  x <- latlon_df_clean(x)
```

## latlon_infer *guess which columns have lat and lon based on aliases like latitude, FacLat, etc.*

### Description

guess which columns have lat and lon based on aliases like latitude, FacLat, etc.

### Usage

```
latlon_infer(mycolnames)
```

### Arguments

| | |
|---|---|
| mycolnames | e.g., colnames(x) where x is a data.frame from read.csv |

### Value

returns all of mycolnames except replacing the best candidates with lat and lon

### See Also

latlon_df_clean() latlon_infer() latlon_is.valid() latlon_as.numeric()

### Examples

```
latlon_infer(c('trilat', 'belong', 'belong')) # warns if no alias found. Does not warn of dupes in other terms
latlon_infer(c('a', 'LONG', 'Longitude', 'lat')) # only the best alias is converted/used
latlon_infer(c('a', 'LONGITUDE', 'Long', 'Lat')) # only the best alias is converted/used
latlon_infer(c('a', 'longing', 'Lat', 'lat', 'LAT')) # case variants of preferred are left alone only if lower
latlon_infer(c('LONG', 'long', 'lat')) # case variants of a single alias are converted to preferred word (if pr
latlon_infer(c('LONG', 'LONG')) # dupes of an alias are renamed and still are dupes! warn!
latlon_infer(c('lat', 'lat', 'Lon')) # dupes left as dupes but warn!
```

## latlon_is.valid *Validate latitudes and longitudes*

### Description

Check each latitude and longitude value to see if they are NA or outside expected numeric ranges (based on approx ranges of lat lon seen among block internal points dataset) lat must be between 17.5 and 71.5, and lon must be ( between -180 and -65) OR (between 172 and 180)

### Usage

```
latlon_is.valid(lat, lon)
```

### Arguments

| | |
|---|---|
| lat | vector of latitudes in decimal degrees |
| lon | numeric vector of longitudes in decimal degrees, same length |

### Value

logical vector, one element per lat lon pair (location)

### See Also

latlon_df_clean() latlon_infer() latlon_is.valid() latlon_as.numeric()

### Examples

```
 ## Not run:
table(latlon_is.valid(lat = EJAMblockdata::blockpoints$lat, lon = EJAMblockdata::blockpoints$lon))
 ##      TRUE
 ## 8,174,955

## End(Not run)
```

---

## latlon_readclean *read csv or xlsx and then clean it This function just wraps* EJAMbatch.summarizer::read_csv_or_xl() *and* latlon_df_clean()

---

### Description

read csv or xlsx and then clean it This function just wraps EJAMbatch.summarizer::read_csv_or_xl() and latlon_df_clean()

### Usage

```
latlon_readclean(x)
```

### Arguments

| | |
|---|---|
| x | character string, full path to a .csv or .xlsx file |

### Value

see latlon_df_clean()

---

## lookup_pctile *Find approx wtd percentiles in lookup table that is in memory*

---

### Description

This is used with a data.frame that is a lookup table used to convert a raw indicator value to a percentile - US, Region, or State percentile.

### Usage

```
lookup_pctile(myvector, varname.in.lookup.table, lookup = usastats, zone)
```

### Arguments

| | |
|---|---|
| myvector | Numeric vector, required. Values to look for in the lookup table. |
| varname.in.lookup.table | Character element, required. Name of column in lookup table to look in to find interval where a given element of myvector values is. |
| lookup | Either lookup must be specified, or a lookup table called us must already be in memory. This is the lookup table data.frame with a PCTILE column and column whose name is the value of varname.in.lookup.table |
| zone | Character element (or vector as long as myvector), optional. If specified, must appear in a column called REGION within the lookup table. For example, it could be 'NY' for New York State. |

### Details

This could be recoded to be more efficient. The data.frame lookup table must have a field called "PCTILE" that has quantiles/percentiles and other column(s) with values that fall at those percentiles. EJAM::usastats, EJAM::statstats, EJAM::regionstats are such lookup tables. This function accepts lookup table (or uses one called us if that is in memory), and finds the number in the PCTILE column that corresponds to where a specified value (in myvector) appears in the column called varname.in.lookup.table. The function just looks for where the specified value fits between values in the lookup table and returns the approximate percentile as found in the PCTILE column. If the value is between the cutpoints listed as percentiles 89 and 90, it returns 89, for example. If the value is exactly equal to the cutpoint listed as percentile 90, it returns percentile 90. If the value is less than the cutpoint listed as percentile 0, which should be the minimum value in the dataset, it still returns 0 as the percentile, but with a warning that the value checked was less than the minimum in the dataset.

### Value

By default, returns numeric vector length of myvector.

---

## merge_state_shapefiles

*OBSOLETE Spatial overlay of facilities points and US States shapefiles to add STATE as column in facility table*

---

### Description

OBSOLETE Spatial overlay of facilities points and US States shapefiles to add STATE as column in facility table

### Usage

```
merge_state_shapefiles(facs, shapefile)
```

### Arguments

| | |
|---|---|
| facs | facilities LONG LAT table |
| shapefile | shapefile of States |

## Value

Returns the facs that was passed to this function, but with a new column, STATE, that has the name of the State each point is inside of

---

`metadata_add` *helper function for package to set attributes of a dataset*

---

## Description

This can be used annually to update some datasets in a package. It just makes it easier to set a few metadata attributes similarly for a number of data elements, for example, to add new or update existing attributes.

## Usage

```
metadata_add(x, metadata)
```

## Arguments

| | |
|---|---|
| `x` | dataset (or any object) whose metadata you want to update or create |
| `metadata` | must be a named list, so that the function can do this for each i: attr(x, which=names(metadata)[i]) <- metadata[i] |

## Value

returns x but with new or altered attributes

## See Also

metadata_check()

## Examples

```
x <- data.frame(a=1:10,b=1001:1010)
metadata <- list(
census_version = 2020,
acs_version = '2016-2020',
acs_releasedate = '3/17/2022',
ejscreen_version = '2.1',
ejscreen_releasedate = 'October 2022',
ejscreen_pkg_data = 'bg22'
)
x <- metadata_add(x, metadata)
attributes(x)
x <- metadata_add(x, list(status='final'))
attr(x,'status')
```

## metadata_check *helper function in updating the package metadata*

### Description

Quick and dirty helper during development, to check all the attributes of all the data files in relevant packages. It loads unloaded packages as needed, which you might not want it to do, but it is not coded to be able to check attributes without doing that.

### Usage

```
metadata_check(
  packages = NULL,
  which = c("census_version", "acs_version", "acs_releasedate", "ACS",
   "ejscreen_version", "ejscreen_releasedate", "ejscreen_pkg_data", "year", "released"),
  loadifnotloaded = TRUE
)
```

### Arguments

| | |
|---|---|
| packages | Optional. e.g. 'EJAMejscreendata', or can be a vector of character strings, and if not specified, default is to report on all packages with EJ as part of their name, like EJAMblockdata or ejscreenapi |
| which | Optional vector (not list) of strings, the attributes. Default is some typical ones used in EJAM-related packages currently. |
| loadifnotloaded | Optional to control if func should temporarily attach packages not already loaded. |

## NAICS *NAICS (industry classification system codes)*

### Description

NAICS (industry classification system codes)

### Details

These industry names and codes get updated every 5 years (2017 version replaced by 2022 version in January 2022). See https://www.census.gov/naics/

```
 This is a list (but may change to data.frame) of NAICS codes.
 The codes are numeric, where names are the code followed by the title of the industrial sector.
  To check the vintage of the dataset, check
   attr(NAICS, 'year')
 The format is like this, and for 2017 version it had 2193 entries:
 but placeholders added in 2023 for 31,32,33, 44,45, 48,49
#  x <- list(
#   `11 - Agriculture, Forestry, Fishing and Hunting` = 11,
#   `111 - Crop Production` = 111,
```

```
#   `1111 - Oilseed and Grain Farming` = 1111,
#   `11111 - Soybean Farming` = 11111,
#   `111110 - Soybean Farming` = 111110
#   )
About NAICS codes:

The North American Industry Classification System (NAICS)
is a system for classifying establishments (individual business locations)
by type of economic activity.
https://www.census.gov/naics/

The North American Industry Classification System (NAICS) is
the standard used by Federal statistical agencies in
classifying business establishments for the purpose of
collecting, analyzing, and publishing statistical data
related to the U.S. business economy.

The codes were updated 2007, 2012, 2017, and for 2022 (announced Dec. 2021).
Finalized changes: https://www.census.gov/naics/federal_register_notices/notices/fr21dc21.pdf
Effective Date for 2022 NAICS
United States codes and Statistical
Policy Directives: Federal statistical
establishment data published for
reference years beginning on or after
January 1, 2022, should be published
using the 2022 NAICS United States
codes. Publication of NAICS United
States, 2022 Manual is planned for
January 2022 on the NAICS website at
www.census.gov/naics. The updated
Statistical Policy Directive No. 8, North
American Industry Classification
System: Classification of
Establishments, will be effective
immediately and will be posted on the
OMB Statistical Programs and Standards
website at www.whitehouse.gov/omb/
  information-regulatory-affairs/
  statistical-programs-standards/.


 ## see   https://www.census.gov/naics/

 # to get 2017 version into this format, see [naics_download()]
 NAICS <- naics_download()
 # specify metadata here on vintage of data, etc.
 usethis::use_data(NAICS, overwrite=TRUE)
 # save(NAICS, file = 'yourpath/EJAM/data/NAICS.rda')
```

## See Also

naics_find() naics_categories() naics_download()

`naics2children` *See NAICS codes queried plus all children of any of those Used by naics_find()*

## Description

See NAICS codes queried plus all children of any of those Used by naics_find()

## Usage

```
naics2children(codes, allcodes = EJAM::NAICS)
```

## Arguments

| | |
|---|---|
| `codes` | vector of numerical or character |
| `allcodes` | Optional (already loaded with package) - dataset with all the codes |

## Details

start with shortest (highest level) codes. since tied for nchar, these branches have zero overlap, so do each. for each of those, get its children = all rows where parentcode == substr(allcodes, 1, nchar(parentcode)) put together list of all codes we want to include so far. now for the next longest set of codes in original list of codes, do same thing. etc. until did it for 5 digit ones to get 6digit children. take the unique(allthat) table(nchar(as.character(NAICS))) 2 3 4 5 6 17 99 311 709 1057

## Value

vector of codes and their names

## See Also

naics_find() NAICS

## Examples

```
naics2children(211)
naics_find(211, exactnumber=TRUE)
naics_find(211, exactnumber=TRUE, add_children = TRUE)
NAICS[211][1:3] # wrong
NAICS[NAICS == 211]
NAICS["211 - Oil and Gas Extraction"]
```

## naics_categories — *See the names of industrial categories and their NAICS code Easy way to list the 2-digit NAICS (17 categories), or other level*

### Description

See the names of industrial categories and their NAICS code Easy way to list the 2-digit NAICS (17 categories), or other level

### Usage

```
naics_categories(digits = 2, dataset = EJAM::NAICS)
```

### Arguments

| | |
|---|---|
| digits | default is 2, for 2-digits NAICS, the top level, but could be up to 6. |
| dataset | Should default to the dataset called NAICS, installed with this package. see NAICS Check attr(NAICS, 'year') |

### Details

Also see https://www.naics.com/search/ There are this many NAICS codes roughly by number of digits in the code: table(nchar(NAICS)) 2 3 4 5 6 17 99 311 709 1057 See https://www.census.gov/naics/

### See Also

naics_find NAICS

### Examples

```
naics_categories()
```

## naics_download — *script to download NAICS file with code and name of sector*

### Description

See source code. Mostly just a short script to get the 2017 or 2022 codes and names. See <'https://www.census.gov/naics/

### Usage

```
naics_download(
  year = 2017,
 urlpattern = "https://www.census.gov/naics/YYYYNAICS/2-6%20digit_YYYY_Codes.xlsx",
  destfile = paste0("~/Downloads/", year, "NAICS.xlsx")
)
```

### Arguments

| | |
|---|---|
| year | which vintage of NAICS codes to use, 2012, 2017, or 2022 |
| urlpattern | full url of xlsx file to use, but with YYYY instead of year |
| destfile | full path and name of file to save as locally |

### Value

names list with year as an attribute

---

## naics_find *Search for an industrial sector in the list of NAICS codes, see subsectors*

---

### Description

Just a utility, quick way to view NAICS industrial sectors that contain queried word or phrase, but can also see all the subcategories within the matching one.

### Usage

```
naics_find(
  query,
  add_children = FALSE,
  naics_dataset = NULL,
  ignore.case = TRUE,
  exactnumber = FALSE,
  search_on_naics_website = FALSE
)
```

### Arguments

| | |
|---|---|
| query | a single word or phrase such as "chemical manufacturing" or "cement" |
| add_children | default is FALSE, so it does NOT chidren (subcategories) of those that match the query. |
| naics_dataset | Should default to the dataset NAICS, installed with this package. see NAICS |
| ignore.case | default TRUE, ignoring whether query is upper or lower case |
| exactnumber | if TRUE, only return the exact match to (each) queried number (NAICS code) |
| search_on_naics_website | if TRUE (not default), returns URL of webpage at naics.com with info on the sector |

### Details

See https://www.census.gov/naics/ NOTE: By default, this shows NAICS that match the text query, and also can include all the children NAICS even if they do not match based on text query. So it first finds NAICS that match the text (or code) query via grep(), and then can also include all subcategories within those categories.

So naics_find('soap', add_children=TRUE) shows "325612 - Polish and Other Sanitation Good Manufacturing", and others, not just "3256 - Soap, Cleaning Compound, and Toilet Preparation Manufacturing", because 3256 matches 'soap' and 325612 is a subcategory of 3256.

The format of NAICS, the naics_dataset, in this package is dput(NAICS1:4) c(11 - Agriculture, Forestry, Fishing = 11, 111 - Crop Production = 111, 1111 - Oilseed and Grain Farming = 1111, 11111 - Soybean Farming = 11111 )

## See Also

naics_categories NAICS naics_findwebscrape() get_facility_info_via_ECHO function naics_url_of_code() naics_url_of_query()

## Examples

```
naics_find(8111, exactnumber = FALSE)
naics_find(8111, exactnumber = TRUE)
naics_find(8111, exactnumber = TRUE, add_children = TRUE)

naics_find("paper")
naics_find("cement | concrete")
cbind(naics_find("pig")
naics_find("pulp", add_children = FALSE)
naics_find("pulp", add_children = TRUE)
naics_find("asdfasdf", add_children = TRUE)
naics_find("asdfasdf", add_children = FALSE)
naics_find("copper smelting", search_on_naics_website=FALSE)
naics_find("copper smelting", search_on_naics_website=TRUE)
# browseURL(naics_find("copper smelting", search_on_naics_website=TRUE))

EJAMfrsdata::frs[EJAMfrsdata::frs$REGISTRY_ID %in% unlist(
  EJAMfrsdata::get_siteid_from_naics(
    EJAM::naics_find("pulp", add_children = TRUE))[,"REGISTRY_ID"]), 1:5]

EJAMejscreenapi::mapfast(EJAMfrsdata::frs[EJAMfrsdata::frs$REGISTRY_ID %in% unlist(
  EJAMfrsdata::get_siteid_from_naics(EJAM::naics_find("pulp"))[,"REGISTRY_ID"]),   ])

 naics_find(211, exactnumber=TRUE)
 naics_find(211, exactnumber=TRUE, add_children = TRUE)
 naics2children(211)
 NAICS[211][1:3] # wrong
 NAICS[NAICS == 211]
 NAICS["211 - Oil and Gas Extraction"]
```

---

`naics_findwebscrape` *for query term, show list of roughly matching NAICS, scraped from web This finds more than just naics_find() does, since that needs an exact match but this looks at naics.com website which lists various aliases for a sector.*

---

## Description

for query term, show list of roughly matching NAICS, scraped from web This finds more than just naics_find() does, since that needs an exact match but this looks at naics.com website which lists various aliases for a sector.

## Usage

```
naics_findwebscrape(query)
```

## Arguments

| | |
|---|---|
| query | text like "gasoline" or "copper smelting" |

## Value

data.frame of info on what was found, naics and title

## See Also

naics_find() naics_url_of_query()

## Examples

```
 naics_find("copper smelting", search_on_naics_website=FALSE)
 naics_find("copper smelting", search_on_naics_website=TRUE)
  naics_url_of_query("copper smelting")
  ## Not run:
  naics_findwebscrape("copper smelting")
  browseURL(naics_url_of_query("copper smelting"))
  browseURL(naics_url_of_code(326))

## End(Not run)
```

---

# naics_url_of_code *Get URL for page with info about industry sector(s) by NAICS See naics.com for more information on NAICS codes*

---

## Description

Get URL for page with info about industry sector(s) by NAICS See naics.com for more information on NAICS codes

## Usage

```
naics_url_of_code(naics)
```

## Arguments

| | |
|---|---|
| naics | vector of one or more NAICS codes, like 11,"31-33",325 |

## Value

vector of URLs as strings like https://www.naics.com/six-digit-naics/?v=2017&code=22

`naics_url_of_query` *Get URL for page with info about industry sectors by text query term See naics.com for more information on NAICS codes*

## Description

Get URL for page with info about industry sectors by text query term See naics.com for more information on NAICS codes

## Usage

```
naics_url_of_query(query)
```

## Arguments

| | |
|---|---|
| `query` | string query term like "gasoline" or "copper smelting" |

## Value

URL as string

`popweightedsums` *Get population weighted sums of indicators*

## Description

Get population weighted sums of indicators

## Usage

```
popweightedsums(data, fieldnames, fieldnames_out, scaling, popname = "POP100")
```

## Arguments

| | |
|---|---|
| `data` | data.table with demographic and/or environmental data |
| `fieldnames` | vector of terms like pctmin, traffic.score, pm, etc. |
| `fieldnames_out` | optional, should be same length as fieldnames |
| `scaling` | number to multiply raw values by to put in right units like percent 0-100 vs 0.0-1.0 |
| `popname` | name of column with population counts to use for weighting |

`proxistat2` *Calculate a proximity score for every blockgroup Indicator of proximity of each blockgroups to some set of facilities or sites. Proximity score is sum of (1/d) where each d is distance of a given site in km, summed over all sites within 5km, as in EJScreen. *** Still need area of each block to fix this func proxistat2()*

## Description

Calculate a proximity score for every blockgroup Indicator of proximity of each blockgroups to some set of facilities or sites. Proximity score is sum of (1/d) where each d is distance of a given site in km, summed over all sites within 5km, as in EJScreen.

*** Still need area of each block to fix this func proxistat2()

## Usage

```
proxistat2(pts, cutoff = 8.04672, quadtree)
```

## Arguments

| | |
|---|---|
| `pts` | data.table of lat lon |
| `cutoff` | distance max, in miles, default is 5km (8.04672 miles) which is the EJScreen max search range for proximity scores |
| `quadtree` | must be localtree from EJAM:: |

## Value

data.table with proximityscore, bgfips, lat, lon, etc.

## Examples

```
 # pts <- testpoints_50
 # x <- proxistat2(pts = pts[1:1000,], quadtree = localtree)
 #
 # summary(x$proximityscore)
 # # analyze.stuff::pctiles(x$proximityscore)
 # plot(x$lon, x$lat)
# tops = x$proximityscore > 500 & !is.infinite(x$proximityscore) & !is.na(x$proximityscore)
 # points(x$lon[tops], x$lat[tops], col="red")
```

`regionstats` *data.table of 100 percentiles and means for each EPA Region.*

## Description

data.table of 100 percentiles and means for each EPA Region (> 1,000 rows) for all the block groups in that zone (e.g., block groups in blockgroupstats) for a set of indicators such as percent low income. Each column is one indicator (or specifies the percentile).

This should be similar to the lookup tables in the gdb on the FTP site of EJScreen.

run_app *Run the Shiny Application*

## Description

Allows package to be a Shiny app and package at the same time.

## Usage

```
run_app(
  onStart = NULL,
  options = list(),
  enableBookmarking = "server",
  uiPattern = "/",
  ...
)
```

## Arguments

| | |
|---|---|
| onStart | A function that will be called before the app is actually run. This is only needed for shinyAppObj, since in the shinyAppDir case, a global.R file can be used for this purpose. |
| options | Named options that should be passed to the runApp call (these can be any of the following: "port", "launch.browser", "host", "quiet", "display.mode" and "test.mode"). You can also specify width and height parameters which provide a hint to the embedding environment about the ideal height/width for the app. |
| enableBookmarking | Can be one of "url", "server", or "disable". The default value, NULL, will respect the setting from any previous calls to enableBookmarking(). See enableBookmarking() for more information on bookmarking your app. |
| uiPattern | A regular expression that will be applied to each GET request to determine whether the ui should be used to handle the request. Note that the entire request path must match the regular expression in order for the match to be considered successful. |
| ... | arguments to pass to golem_opts. See ?golem::get_golem_options for more details. |

## Details

Normally R Shiny apps are not R packages - The server just sources all .R files found in the /R/ folder, and then runs what is found in app.R (if that is found / it is a one-file Shiny app). This R Shiny app, however, is shared as an R package, via the golem package approach, which provides the useful features of a package and useful features that the golem package enables.

There is still an app.R script in the package root – note there is no function called app() – which lets RStudio Connect source the app.R script to launch this shiny app.

The way this works is that there is a file called

_disable_autoload.R in the /R/ folder

to tell the server to not source all the source .R files, since they are already in the installed package. Then they get loaded from the package because the app.R script here says this:

```
pkgload::load_all(export_all = FALSE,helpers = FALSE,attach_testthat = FALSE)
```

with the shinyApp() call wrapped in shiny::runApp() rather than in app()

Also, app_runYYYY() is the same as YYYY::run_app() in case that is useful.

See https://thinkr-open.github.io/golem/

---

`sitepoints_example` *data.table of points as example of sitepoints for EJAM*

---

## Description

data.table of points as example of sitepoints for EJAM

---

`sites2blocks_example` *data.table of output of* `getblocknearby()`*, each row is a unique site-block-distance*

---

## Description

data.table of output of `getblocknearby()`, each row is a unique site-block-distance

---

`stateinfo` *data.frame of state abbreviations and state names (50+DC+PR; not AS, GU, MP, VI, UM)*

---

## Description

52 rows and 5 variables: ST is the 2-letter abbreviation, statename is the State name (and ftpname is the name as used on Census FTP site).

## Details

column names: "ST" "statename" "ftpname" "FIPS.ST" "REGION"

Some datasets lack PR. (72) Many datasets lack these: AS, GU, MP, VI (codes "60" "66" "69" "78") Almost all datasets lack UM. (74)

```
72 PR                 Puerto Rico
66 GU                        Guam
69 MP    Northern Mariana Islands
78 VI         U.S. Virgin Islands
74 UM U.S. Minor Outlying Islands
```

stateinfo <- structure(list( ST = c("AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DE", "DC", "FL", "GA", "HI", "ID", "IL", "IN", "IA", "KS", "KY", "LA", "ME", "MD", "MA", "MI", "MN", "MS", "MO", "MT", "NE", "NV", "NH", "NJ", "NM", "NY", "NC", "ND", "OH", "OK", "OR", "PA", "RI", "SC", "SD", "TN", "TX", "UT", "VT", "VA", "WA", "WV", "WI", "WY", # "AS", "GU", "MP","VI" # "UM", #### U.S. Minor Outlying Islands # "US", "PR"),

statename = c("Alabama", "Alaska", "Arizona", "Arkansas", "California", "Colorado", "Connecticut", "Delaware", "District of Columbia", "Florida", "Georgia", "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland", "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey", "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio", "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina", "South Dakota", "Tennessee", "Texas", "Utah", "Vermont", "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming", # "American Samoa", "Guam", "Northern Mariana Islands", "U.S. Virgin Islands", # "U.S. Minor Outlying Islands", # "United States", "Puerto Rico"),

ftpname = c("Alabama", "Alaska", "Arizona", "Arkansas", "California", "Colorado", "Connecticut", "Delaware", "DistrictOfColumbia", "Florida", "Georgia", "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland", "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana", "Nebraska", "Nevada", "NewHampshire", "NewJersey", "NewMexico", "NewYork", "NorthCarolina", "NorthDakota", "Ohio", "Oklahoma", "Oregon", "Pennsylvania", "RhodeIsland", "SouthCarolina", "SouthDakota", "Tennessee", "Texas", "Utah", "Vermont", "Virginia", "Washington", "WestVirginia", "Wisconsin", "Wyoming", # NA, NA, NA, NA, # NA, #### U.S. Minor Outlying Islands # "UnitedStates", "PuertoRico"),

FIPS.ST = c("01", "02", "04", "05", "06", "08", "09", "10", "11", "12", "13", "15", "16", "17", "18", "19", "20", "21", "22", "23", "24", "25", "26", "27", "28", "29", "30", "31", "32", "33", "34", "35", "36", "37", "38", "39", "40", "41", "42", "44", "45", "46", "47", "48", "49", "50", "51", "53", "54", "55", "56", # "60", "66", "69", "78", # "74", #### U.S. Minor Outlying Islands # NA, #### US "72"), REGION = c(4, 10, 9, 6, 9, 8, 1, 3, 3, 4, 4, 9, 10, 5, 5, 7, 7, 4, 6, 1, 3, 1, 5, 5, 4, 7, 8, 7, 9, 1, 2, 6, 2, 4, 8, 5, 6, 10, 3, 1, 4, 8, 4, 6, 8, 1, 3, 10, 3, 5, 8, # NA, NA, NA, NA, # NA, ##### U.S. Minor Outlying Islands # NA, # US 2) ), row.names = c(NA, -52L), class = "data.frame")

---

`statestats` *data.table of 100 percentiles and means for each US State and PR and DC.*

---

## Description

data.table of 100 percentiles and means for each US State and PR and DC (approx 5,300 rows) for all the block groups in that zone (e.g., block groups in blockgroupstats) for a set of indicators such as percent low income. Each column is one indicator (or specifies the percentile).

This should be similar to the lookup tables in the gdb on the FTP site of EJScreen.

---

`summarize_blockcount` *Get summary stats on counts of blocks (unique vs doublecounted) near sites*

---

## Description

Get summary stats on counts of blocks (unique vs doublecounted) near sites

### Usage

```
summarize_blockcount(x)
```

### Arguments

| | |
|---|---|
| x | The output of `getblocksnearby()` like sites2blocks_example |

### Value

A list of stats

### Examples

```
summarize_blockcount(sites2blocks_example)
```

---

## summarize_blocks_per_site

*Get summary stats on counts of blocks near various sites*

---

### Description

Tells you # of blocks near avg site, how many sites have only 1 block nearby, or have <30 nearby, etc.

### Usage

```
summarize_blocks_per_site(x, varname = "siteid")
```

### Arguments

| | |
|---|---|
| x | The output of getblocksnearby() |
| varname | colname of variable in data.table x that is the one to summarize by |

### Value

invisibly, a list of stats

## summarize_sites_per_block

*Get summary stats on how many sites are near various blocks (residents)*

### Description

Get summary stats on how many sites are near various blocks (residents)

### Usage

```
summarize_sites_per_block(x, varname = "blockid")
```

### Arguments

| | |
|---|---|
| x | The output of `getblocksnearby()` like sites2blocks_example |
| varname | colname of variable in data.table x that is the one to summarize by |

### Value

invisibly, a list of stats

## testpoints_1000_dt

*Random test points data.table with columns lat lon site*

### Description

Random test points data.table with columns lat lon site

## testpoints_100_dt

*Random test points data.table with columns lat lon site*

### Description

Random test points data.table with columns lat lon site

## testpoints_blockpoints *Get some random US locations as points to try out/ for testing*

### Description

Get some random US locations as points to try out/ for testing

### Usage

```
testpoints_blockpoints(n = 10, weighting = "geo", ST = is.null, as.dt = TRUE)
```

### Arguments

| | |
|---|---|
| n | how many points do you want? |
| weighting | geo means each block is equally likely, pop means the points are population weighted (Census 2020 pop) so they represent a random sample of where US residents live - the average person. |
| ST | can be a character vector of 2 letter State abbreviations to pick from only some States |
| as.dt | if TRUE (default), a data.table, but if FALSE then a data.frame |

### Value

see as.dt paramter. It returns a table with columns blockid, lat, lon

## usastats *data.table of 100 percentiles and means*

### Description

data.table of 100 percentiles and means (about 100 rows) in the USA overall, across all locations (e.g., block groups in blockgroupstats) for a set of indicators such as percent low income. Each column is one indicator (or specifies the percentile).

This should be similar to the lookup tables in the gdb on the FTP site of EJScreen.

`write_pctiles_lookup` *create lookup table of percentiles 0 to 100 and mean for each indicator by State or USA total*

## Description

create lookup table of percentiles 0 to 100 and mean for each indicator by State or USA total

## Usage

```
write_pctiles_lookup(
  x,
  zone.vector = NULL,
  zoneOverallName = "USA",
  wts = NULL
)
```

## Arguments

| | |
|---|---|
| `x` | data.frame with numeric data. Each column will be examined to calculate mean, sd, and percentiles, for each zone |
| `zone.vector` | optional names of states or regions, for example. same length as wts, or rows in mydf |
| `zoneOverallName` | optional. Default is USA. |
| `wts` | not used in EJScreen percentiles anymore |

# Index