Package ‘EJAM’

April 19, 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),
EJAMblockdata,
EJAMfrsdata,
EJAMbatch.summarizer,
EJAMejscreenapi

Imports attempt,
collapse,
config (>= 0.3.1),
data.table,
DBI,
doSNOW,
DT,
EJAMejscreendata,
foreach,
ggplot2,
glue,
golem (>= 0.3.3),
htmltools,
leaflet,
magrittr,
openxlsx,
pdist,
pkgload,
readxl,

rmarkdown,
RMySQL,
SearchTrees,
shinyBS,
shinycssloaders,
shinyjs,
sf,
sp,
tidyverse,
dplyr,
tidyr

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

Suggests knitr,
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` *Launch the Shiny Application in RStudio*

Description

launch Shiny web app from RStudio

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.

Details

app_run_EJAM() is like `run_app()` app_run_EJAMejscreenapi() is like `EJAMejscreenapi::run_app()`

`app_server` *EJAM app server*

Description

EJAM app server

Usage

```
app_server(input, output, session)
```

Arguments

`input, output, session`	Internal parameters for shiny. DO NOT REMOVE.

`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 i
 # 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
```

bg_cenpop2020 — *data.table with all US Census 2020 block groups, Census 2020 population count, and lat/lon of Census2020-population-weighted centroid of block group*

Description

data.table with all US Census 2020 block groups, Census 2020 population count, and lat/lon of Census2020-population-weighted centroid of block group

Details

also see attributes(bg_cenpop2020) for source URL and date

See Also

bgpts blockgroupstats

`bg_from_county` *Analyze US Counties as if they were sites, to get EJ indicators summary for each county*

Description

Analyze US Counties as if they were sites, to get EJ indicators summary for each county

Usage

```
bg_from_county(fips)
```

Arguments

`fips`	County FIPS vector as character not numeric values

Value

data.table with all pairs of county fips - bgid, and a unique siteid assigned to each county

Examples

```
bg_from_county(c('01001','72153'))
# Largest US Counties by ACS Population Totals:
blockgroupstats[ , .(ST = ST[1], countypop = sum(pop)),
 by=.(FIPS = substr(bgfips,1,5))][order(-countypop),][1:20, .(
 CountyPopulation = prettyNum(countypop, big.mark = ","), FIPS, ST)]
```

`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

---

`counties_as_sites` *Analyze US Counties as if they were sites, to get EJ indicators summary for each county*

Description

Analyze US Counties as if they were sites, to get EJ indicators summary for each county

Usage

```
counties_as_sites(fips)
```

Arguments

`fips`	County FIPS vector as character not numeric values

Value

data.table with all pairs of county fips - bgid, and a unique siteid assigned to each county

Examples

```
bg_from_county(c('01001','72153'))
# Largest US Counties by ACS Population Totals:
blockgroupstats[ , .(ST = ST[1], countypop = sum(pop)),
 by=.(FIPS = substr(bgfips,1,5))][order(-countypop),][1:20, .(
 CountyPopulation = prettyNum(countypop, big.mark = ","), FIPS, ST)]
```

`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)
```

`distance_by_group` *distance_by_group Get average distance for ONE demographic group versus everyone else*

Description

distance_by_group Get average distance for ONE demographic group versus everyone else

Usage

```
distance_by_group(
  results_bybg_people,
  demogvarname = "Demog.Index",
  demoglabel = demogvarname
)
```

Arguments

`results_bybg_people`	data.table from doaggregate()$results_bybg_people
`demogvarname`	e.g., "pctlowinc"
`demoglabel`	e.g., "Low Income Residents"

Details

Note on Avg Distance and range of distances in each Demog group, & %D as function of distance:

We have info on each blockgroup near each site, which means some small % of those bgs are duplicated in this table:

```
results_bybg_people
```

Mostly we want overall (not by site) to know avg and cum distrib of distances in each demog,

(and also %D as a function of continuous distance),

and for those stats we would want to take only unique blockgroups from here, using the shorter distance, so the distribution of distances does not doublecount people.

But we might also want to see that distribution of distances by D for just 1 site?

And we might also want to see the %D as a function of continuous distance at just 1 site?

So to retain flexibility doaggregate() reports all instances of blockgroup-site pairings.

Value

list of 2 numbers: avg_distance_for_group and avg_distance_for_nongroup

See Also

`distance_by_group_plot()` `distance_by_groups()`

`distance_cdf_by_group_plot`

distance_cdf_by_group_plot - SLOW - needs to be optimized Plot a graphic showing cumulative shares of ONE demographic group that are within each distance

---

Description

distance_cdf_by_group_plot - SLOW - needs to be optimized Plot a graphic showing cumulative shares of ONE demographic group that are within each distance

Usage

```
distance_cdf_by_group_plot(
  results_bybg_people,
  radius_miles = round(max(x$distance_min_avgperson, na.rm = T), 1),
  demogvarname = "Demog.Index",
  demoglabel = demogvarname,
  color1 = "red",
  color2 = "black"
)
```

Arguments

`results_bybg_people`	data.table from doaggregate()$results_bybg_people
`radius_miles`	miles radius that was max distance analyzed
`demogvarname`	name of column in results_bybg_people, e.g., "pctlowinc"
`demoglabel`	friendly text name for labelling graphic, like "Low income residents"
`color1`	color like "red" for demographic group of interest
`color2`	color like "gray" for everyone else

Value

invisibly returns full table of sorted distances of blockgroups, cumulative count of demog group at that block group's distance, and cumulative count of everyone else in that block group

See Also

`distance_by_group()` `getblocksnearbyviaQuadTree()` for examples

`distance_via_surfacedistance`

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

```
distance_via_surfacedistance(x)
```

Arguments

`x`	surface distance in miles

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

Description

This updated 2023 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,
  sites2states_or_latlon = NA,
  countcols = NULL,
  popmeancols = NULL,
  calculatedcols = NULL,
  testing = FALSE,
  include_ejindexes = FALSE,
  updateProgress = NULL,
  need_proximityscore = FALSE,
  silentinteractive = 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 get-blocksnearby function. See sites2blocks_example dataset in package, as input to this function
`sites2states_or_latlon`	data.table or just data.frame, with columns siteid (each unique one in sites2blocks) and ST (2-character State abbreviation) or lat and lon
`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
`include_ejindexes`	not yet implemented
`updateProgress`	progress bar function used for shiny app
`need_proximityscore`	whether to calculate proximity scores

`silentinteractive`	Set to FALSE to prevent long output showing in console in RStudio when in interactive mode
`...`	more to pass to another function? Not used currently.

Details

For all examples, see `getblocksnearbyviaQuadTree()`

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
- Index built from quaddata, and blockquadtree: data.table and quad tree, for indexes of block points (and localtree that is created when package is loaded)
- EJAM::blockgroupstats - A data.table (such as EJScreen demographic and environmental data by blockgroup?)

See Also

ejamit `getblocksnearby()`

`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()`

`ejamit` *Get complete EJ analysis numbers (demographic and environmental indicators) near a list of locations*

Description

This is the main function in EJAM for users who want to use EJAM from RStudio. It does essentially what the webapp does to analyze/summarize near a set of points. See help("EJAM")

Usage

```
ejamit(
  sitepoints,
  cutoff = 3,
  maxcutoff = 31.07,
  avoidorphans = TRUE,
  quadtree = NULL,
  ...
)
```

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.

`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 is automatically created when the package is loaded via the `.onLoad()` function
`...`	passed to `getblocksnearbyviaQuadTree()` or other such functions

Value

A list of tables of results.

See Also

`getblocksnearby()` `doaggregate()`

Examples

```
# All in one step, using functions not shiny app:
## Not run:
out <- ejamit(testpoints_100_dt, 2, quadtree=localtree)

# Do not specify sitepoints and it will prompt you for a file,
# if in RStudio in interactive mode!
out <- ejamit(cutoff = 3)

 # Specify facilities or sites as points for test data,
 # use 1000 test facility points from the R package
 testsites <- testpoints_1000_dt
 # use facility points in an excel or csv file
 testsites <- latlon_from_anything(
  "./inst/testdata/testpoints_207_sites_with_signif_violations_NAICS_326_ECHO.csv")
 # use facility points from a random sample of EPA-regulated facilities
 testsites <- testpoints_n(1e3)

 # Specify max distance from sites to look at (residents within X miles of site point)
 radius <- 3.1 # miles

 # Get summaries of all indicators near a set of points
 out <- ejamit(testsites, radius)
 # out <- ejamit("myfile.xlsx", 3.1)

 # out2 <- EJAMejscreenapi::ejscreenit(testpoints_1000[1:3,])

 # View results overall
 round(t(out$results_overall), 3.1)

 # View plots
 # plot_distance_avg_by_group(out)
 # plot_distance_cdf_by_group(out)

 # View maps
 mapfast(out$results_bysite, radius = 3.1)

 # view results at a single site
 t(out$results_bysite[1, ])
 t(out$results_bysite[out$results_bysite$siteid == 2, ])
```

```
  # if doing just 1st step of ejamit()
  #  get distance between each site and every nearby Census block
  s2b <- testdata_sites2blocks
  s2b <- getblocksnearby(testsites, cutoff = radius)
  s2b <- getblocksnearbyviaQuadTree(testsites, cutoff = radius)
  summarize_blockcount(s2b)

  # if doing just 2d step of ejamit()
  #  get summaries of all indicators based on table of distances
  out <- doaggregate(s2b, testsites) # this works now and is simpler


## End(Not run)
```

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

Description

list of names of EJAM-related R packages

`fipsbg_from_anyfips` *fipsbg_from_anyfips convert any FIPS codes to the FIPS of all the blockgroups that are among or within or containing those FIPS*

Description

fipsbg_from_anyfips convert any FIPS codes to the FIPS of all the blockgroups that are among or within or containing those FIPS

Usage

```
fipsbg_from_anyfips(fips)
```

Arguments

`fips`	vector of US FIPS codes, as character or numeric, with or without their leading zeroes, each with as many characters

Details

This is a way to get a list of blockgroups, specified by state/county/tract or even block.

Takes a vector of one or more FIPS that could be State (2-digit), County (5-digit), Tract (11-digit), or blockgroup (12 digit), or even block (15-digit fips).

Returns unique vector of FIPS of all US blockgroups (including DC and Puerto Rico) that contain any specified blocks, are equal to any specified blockgroup fips, or are contained within any provided tract/county/state FIPS.

stateinfo$STmatch(unique(substr(blockgroupstats$bgfips,1,2)), stateinfo$FIPS.ST)

Value

vector of blockgroup FIPS (or NA values) that may be much longer than the vector of fips passed to this function.

See Also

fips_lead_zero()

Examples

```
# all blockgroups in one state
blockgroupstats[,.N,by=substr(bgfips,1,2)]
length(fipsbg_from_anyfips("72"))
# all blockgroups in this one county
fipsbg_from_anyfips(30001)
# all blockgroups that contain any of these 6 blocks (just one bg)
fipsbg_from_anyfips( blockid2fips$blockfips[1:6])
# 2 counties
fipsbg_from_anyfips(c(36009,36011))
```

`fips_lead_zero` *fips_lead_zero Add leading zeroes to fips codes if missing, replace with NA if length invalid Note it does NOT VALIDATE FIPS - It does NOT check if FIPS is valid other than checking its length seems OK, i.e., it might be a state, county, tract, blockgroup, or block FIPS code.*

Description

fips_lead_zero Add leading zeroes to fips codes if missing, replace with NA if length invalid Note it does NOT VALIDATE FIPS - It does NOT check if FIPS is valid other than checking its length seems OK, i.e., it might be a state, county, tract, blockgroup, or block FIPS code.

Usage

```
fips_lead_zero(fips)
```

Arguments

`fips`	vector of numeric or character US FIPS codes

Value

vector of same length

Examples

```
fips_lead_zero(c(1,"01",1234,"1234","12345",123456))
```

`format_gt_table` *format_gt_table*

Description

format_gt_table

Usage

```
format_gt_table(
  df,
  type,
  my_cell_color = "#dce6f0",
  my_border_color = "#0070c0"
)
```

Arguments

`df,`	a data frame with 6 columns (var_names, value, state_avg, state_pctile, usa_avg, usa_pctile), and one row per indicator
`type,`	string - one of 'demog', 'envt'
`my_cell_color,`	color for filling in background of table cells, can be given as string ('blue') or hex code ('#0070c0')
`my_border_color,`	color for table borders and boundaries, can be given as string ('blue') or hex code ('#0070c0')

Value

a 'gt'-style table with formatting to closely match EJScreen standard report formatting

`frs_from_naics` *Use NAICS code or industry title text search to see FRS facility registry services data on those EPA-regulated sites*

Description

Use NAICS code or industry title text search to see FRS facility registry services data on those EPA-regulated sites

Usage

```
frs_from_naics(naics_code_or_name, ...)
```

Arguments

`naics_code_or_name`	
`...`	passed to `naics_from_any()`

Value

relevant rows of the data.table called frs, which has column names that are "lat" "lon" "REGISTRY_ID" "PRIMARY_NAME" "NAICS" "PGM_SYS_ACRNMS"

See Also

`siteid_from_naics()` `naics_from_any()`

Examples

```
frs_from_naics("uranium")
mapfast(frs_from_naics(naics_from_any("nuclear")$code))
naics_from_any("silver")
naics_from_name("silver")
naics_from_any(212222 )
frs_from_naics(21222)
siteid_from_naics(21222)
latlon_from_naics(21222)
```

`frs_from_program` *Use EPA Program acronym like TRIS to see FRS facility registry services data on those EPA-regulated sites*

Description

Get data.table based on given FRS Program System CATEGORY. Find all FRS sites in a program like RCRAINFO, TRIS, or others.

Usage

```
frs_from_program(program)
```

Arguments

`program`	vector of one or more EPA Program names used by FRS

Value

relevant rows of the data.table called frs, which has column names that are "lat" "lon" "REGISTRY_ID" "PRIMARY_NAME" "NAICS" "PGM_SYS_ACRNMS"

Examples

```
mapfast(latlon_from_program("CAMDBS"))
#p1="TRIS"; p2="EIS";
#analyze.stuff::overlaps(
#  latlon_from_program(p1)$REGISTRY_ID,
#  latlon_from_program(p2)$REGISTRY_ID,
#  ab_names = c(p1,p2)
#)
```

`frs_from_programid` *Use EPA Program ID to see FRS facility registry services data on those EPA-regulated sites*

Description

Use EPA Program ID to see FRS facility registry services data on those EPA-regulated sites

Usage

```
frs_from_programid(programid)
```

Arguments

`siteid`	vector of one or more EPA Program ID codes used by FRS

Value

relevant rows of the data.table called frs, which has column names that are "lat" "lon" "REGISTRY_ID" "PRIMARY_NAME" "NAICS" "PGM_SYS_ACRNMS"

Examples

```
x=frs_from_programid(testids_program_sys_id)
  x
  mapfast(x)
```

`frs_from_regid` *Use registry ID to see FRS facility registry services data on those EPA-regulated sites*

Description

Use registry ID to see FRS facility registry services data on those EPA-regulated sites

Usage

```
frs_from_regid(siteid)
```

Arguments

`siteid`	vector of one or more EPA Registry ID codes used by FRS

Value

relevant rows of the data.table called frs, which has column names that are "lat" "lon" "REGISTRY_ID" "PRIMARY_NAME" "NAICS" "PGM_SYS_ACRNMS"

Examples

```
frs_from_siteid(testids_registry_id)
```

`frs_from_siteid` *Use registry ID to see FRS facility registry services data on those EPA-regulated sites*

Description

Use registry ID to see FRS facility registry services data on those EPA-regulated sites

Usage

```
frs_from_siteid(siteid)
```

Arguments

`siteid`	vector of one or more EPA Registry ID codes used by FRS

Value

relevant rows of the data.table called frs, which has column names that are "lat" "lon" "REGISTRY_ID" "PRIMARY_NAME" "NAICS" "PGM_SYS_ACRNMS"

Examples

```
frs_from_siteid(testids_registry_id)
```

`frs_from_sitename` *Use site name text search to see FRS facility registry services data on those EPA-regulated sites VERY SLOW search within PRIMARY_NAME of facilities for matching text*

Description

Use site name text search to see FRS facility registry services data on those EPA-regulated sites VERY SLOW search within PRIMARY_NAME of facilities for matching text

Usage

```
frs_from_sitename(sitenames, ignore.case = TRUE, fixed = FALSE)
```

Arguments

`sitenames`	one or more strings in a vector, which can be regular expressions or query for exact match using fixed=TRUE
`ignore.case`	logical, search is not case sensitive by default (unlike `grepl()` default)
`fixed`	see `grepl()`, if set to TRUE it looks for only exact matches

Value

relevant rows of the data.table called frs, which has column names that are "lat" "lon" "REGISTRY_ID" "PRIMARY_NAME" "NAICS" "PGM_SYS_ACRNMS"

Examples

```
## Not run:
 # very slow
 x=frs_from_sitename
 nrow(x)
 head(x)

## End(Not run)
```

---

frs_is_valid *Validate FRS Registry ID list uploads*

Description

Check for proper FRS facility id in uploaded data

Usage

```
frs_is_valid(frs_upload)
```

Arguments

`frs_upload`	upload frs registry IDs table converted to data frame

Value

boolean value (valid or not valid)

---

getblocksnearby *Fast way to find nearby points - finds distance to each Census block centroid nearby*

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

```
getblocksnearby(
  sitepoints,
  cutoff = 3,
  maxcutoff = 31.07,
  avoidorphans = TRUE,
  quadtree,
  parallel = FALSE,
  ...
)
```

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.
`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 is automatically created when the package is loaded via the `.onLoad()` function
`...`	passed to `getblocksnearbyviaQuadTree()` or other such functions

Details

See `ejamit()` for examples.

getblocksnearby() is a wrapper redirecting to the right version, like `getblocksnearbyviaQuadTree()` Census block "internal points" (defined by Census Bureau) are actually what it looks for, and they are like centroids. The blocks are pre-indexed for the whole USA, via the data object quadtree aka localtree

See Also

`ejamit()` `getblocksnearbyviaQuadTree()` `getblocksnearbyviaQuadTree_Clustered()` `getblocksnearbyviaQua`

`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 = 3,
  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

For all examples, see `ejamit()`

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_and_doaggregate()` `getblocksnearby()` `getblocksnearbyviaQuadTree()` `getblocksnearbyviaQuadTree_Clustered()` `getblocksnearbyviaQuadTree2()`

`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 = 3,
  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 is automatically created when the package is loaded via the `.onLoad()` function

See Also

`ejamit()` `getblocksnearby()` `getblocksnearbyviaQuadTree()` `getblocksnearbyviaQuadTree_Clustered()` `getblocksnearbyviaQuadTree2()`

`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 = 3,
  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)
```

`getblocksnearbyviaQuadTree3` *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

```
getblocksnearbyviaQuadTree3(
  sitepoints,
  cutoff = 3,
  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 is automatically created when the package is loaded via the `.onLoad()` function

See Also

`ejamit()` `getblocksnearby()` `getblocksnearbyviaQuadTree()` `getblocksnearbyviaQuadTree_Clustered()` `getblocksnearbyviaQuadTree2()`

`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(
  sitepoints,
  cutoff,
  maxcutoff,
  avoidorphans,
  CountCPU = 1,
  quadtree
)
```

Arguments

`sitepoints`	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()
`quadtree`	index of all US blocks like localtree

Details

For all examples, see `getblocksnearbyviaQuadTree()`

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

See Also

`getblocksnearby_and_doaggregate()` `getblocksnearby()` `getblocksnearbyviaQuadTree()` `getblocksnearbyviaQuadTree_Clustered()` `getblocksnearbyviaQuadTree2()`

`get_blockpoints_in_shape` *find blocks that are in a polygon, using internal point of block - WORK IN PROGRESS *****

Description

This is like getblocksnearby() but for a polygonal buffer area instead of a circular buffer.

Usage

```
get_blockpoints_in_shape(
  polys,
  addedbuffermiles = 0,
  blocksnearby = NULL,
  dissolved = FALSE,
  safety_margin_ratio = 1.1
)
```

Arguments

`polys`	Spatial data as from sf::st_as_sf(), with a column called siteid, like points as from `get_shapefile_from_sitepoints()`, or a table of points with lat,lon columns that will first be converted here using that function, or polygons (not yet tested).
`addedbuffermiles`	width of optional buffering to add to the points (or edges), in miles
`blocksnearby`	optional table of blocks with blockid,siteid (from which lat,lon can be looked up in blockpoints dt)
`dissolved`	If TRUE, use sf::st_union(polys) to find unique blocks inside any one or more of polys
`safety_margin_ratio`	multiplied by addedbuffermiles, how far to search for blocks nearby using get-blocksnearby(), before using those found to do the intersection via sf::

Details

This uses getblocksnearby() to get a very fast rough/good estimate of which US block points are nearby (with a safety margin - see param below), before then using sf:: to carefully identify which of those candidate blocks are actually inside each polygon (e.g., circle) according to sf:: methods. For circular buffers, just using getblocksnearby() should work and not need this function. For noncircular polygons, buffered or not, this function will provide a way to very quickly filter down to which of the millions of US blocks should be examined by the sf:: join / intersect, since otherwise it takes forever for sf:: to check all US blocks.

Value

Block points table for those blocks whose internal point is inside the buffer which is just a circular buffer of specified radius if polys are just points.

See Also

`get_blockpoints_in_shape()` `get_shapefile_from_sitepoints()` `get_shape_buffered_from_shapefile_poin`

Examples

```
  x = get_shapefile_from_sitepoints(testpoints_n(2))
  # y = get_blockpoints_in_shape(x, 1)  # very very slow
```

`get_circles_from_spatialpoints`
add buffer around shape (points, here)

Description

add buffer around shape (points, here)

Usage

```
get_circles_from_spatialpoints(shapefile_points, radius.miles, ...)
```

Arguments

`shapefile_points`	spatial object like areas at high risk or areas with facilities to be analyzed
`radius.miles`	width of buffer to add to shapefile_points (in case dist is a units object, it should be convertible to arc_degree if x has geographic coordinates, and to st_crs(x)$units otherwise)
`...`	passed to st_buffer()

Details

Just a wrapper for `sf::st_buffer()`

See Also

`get_blockpoints_in_shape()` `get_shapefile_from_sitepoints()` get_shape_buffered_from_shapefile_points

`get_shapefile_from_sitepoints`

convert table of lat,lon points/sites into sf:: shapefile Creates a simple feature (sf) dataframe from points

---

Description

convert table of lat,lon points/sites into sf:: shapefile Creates a simple feature (sf) dataframe from points

Usage

```
get_shapefile_from_sitepoints(sitepoints)
```

Arguments

`sitepoints`	a data.table or data.frame with columns called lat,lon

Value

A shapefile via `sf::st_as_sf()`

See Also

`get_blockpoints_in_shape()` `get_shapefile_from_sitepoints()` `get_shape_buffered_from_shapefile_poin`

`get_shape_buffered_from_shapefile_points` *add buffer around shape (points, here)*

Description

add buffer around shape (points, here)

Usage

```
get_shape_buffered_from_shapefile_points(shapefile_points, radius.miles, ...)
```

Arguments

`shapefile_points`	spatial object like areas at high risk or areas with facilities to be analyzed
`radius.miles`	width of buffer to add to shapefile_points (in case dist is a units object, it should be convertible to arc_degree if x has geographic coordinates, and to st_crs(x)$units otherwise)
`...`	passed to st_buffer()

Details

Just a wrapper for `sf::st_buffer()`

See Also

`get_blockpoints_in_shape()` `get_shapefile_from_sitepoints()` get_shape_buffered_from_shapefile_points

`input_names_listing` *input_names_listing Utility checking values of input$ that appear in this code*

Description

input_names_listing Utility checking values of input$ that appear in this code

Usage

```
input_names_listing(file = "./R/app_server.R")
```

Arguments

`file`	path to source file to search in

Value

character vector of ids of inputs like x,y,z if it found input$x input$y input$z

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

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

Used by `latlon_from_anything()`. Uses `latlon_infer()` `latlon_is.valid()` `latlon_as.numeric()`

Examples

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

`latlon_from_anything` *Flexibly get lat/lon from file, data.frame, data.table, or lat/lon vectors*

Description

Try to figure out if user provided latitude / longitude as vectors, data.frame, file, or interactively pick file.

Usage

```
latlon_from_anything(x, y)
```

Arguments

x	If missing and interactive mode in RStudio, prompts user for file. Otherwise, this can be a filename (csv or xlsx, with path), or data.frame/ data.table/ matrix, or vector of longitudes (in which case y must be the latitudes). Note that even though it is called latlon_etc the lon is x and comes before the lat among parameters x,y File or data.frame/data.table/matrix must have columns called lon and lat, or something that can be inferred to be that by latlon_infer()
y	If x is a vector of longitudes, y must be the latitudes. Ignored otherwise.

Details

This function, `latlon_from_anything()`

relies on

`EJAMbatch.summarizer::read_csv_or_xl()` = `EJAMejscreenapi::read_csv_or_xl()` and

`latlon_df_clean()` which in turn uses `latlon_infer()` `latlon_as.numeric()` `latlon_is.valid()`

`EJAMejscreenapi::read_and_clean_points()`

would be the most general / flexible broadest way to get points, but is still work in progress

is similar to what is done by `latlon_from_anything()`

except it also uses these functions:

`EJAMfrsdata::latlon_from_siteid()`

`EJAMfrsdata::latlon_from_programid()` but not _from_naics() ?

Value

A data.frame that has at least columns lon and lat (and others if they were in x)

See Also

`EJAMbatch.summarizer::read_csv_or_xl()` `latlon_df_clean()`

Examples

```
if (interactive()) {
pts <- latlon_from_anything()
}
latlon_from_anything(system.file("testdata/Sample12.xlsx",
  package="EJAMejscreenapi"))
latlon_from_anything(system.file("testdata/testpoints_05.csv",
  package="EJAMejscreenapi"))
latlon_from_anything(testpoints_50[1:6,] )
latlon_from_anything(testpoints_50[1:6, c('lat','lon')] )
latlon_from_anything(x=testpoints_50$lon[1:6], y=testpoints_50$lat[1:6] )
```

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,
  #  but doesnt warn of dupes in other terms, just preferred term.
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 lowercase one is found
latlon_infer(c('LONG', 'long', 'lat')) # case variants of a single alias are
    # converted to preferred word (if pref not found), creating dupes!  warn!
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)
```

`lat_alias, lon_alias` *Synonyms for lat and lon*

Description

lists of synonyms for "latitude" and "longitude" used when guessing which column is what in user-provided tables of coordinates

---

`map_facilities` *map_facilities*

Description

make a leaflet map of uploaded points

Usage

```
map_facilities(mypoints, rad = 3, highlight = FALSE, clustered)
```

Arguments

`mypoints,`	data frame of uploaded points
`rad,`	a size for drawing each circle (buffer search radius)
`highlight,`	a logicial for whether to highlight overlapping points (defaults to FALSE)
`clustered,`	a vector of T/F values for each point, indicating if they overlap with another

Value

a leaflet map with circles, circleMarkers, and basic popup

`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` *named list of all NAICS code numbers and industry name for each*

Description

named list of all NAICS code numbers and industry name for each

Details

see `https://naics.com`

See Also

naicstable `naics_from_any()` `naics_categories()` NAICS

`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"]
```

`naicstable` *data.table with all NAICS code numbers and industry name for each*

Description

data.table with all NAICS code numbers and industry name for each

Details

see https://naics.com

See Also

naics_from_any() naics_categories() NAICS naics_findwebscrape()

`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_findwebscrape`	*for query term, show list of roughly matching NAICS, scraped from web This finds more than just `naics_from_any()` 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_from_any()` 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_from_any()` `url_naics.com()`

Examples

```
 # naics_from_any("copper smelting")
 # naics_from_any("copper smelting", website_scrape=TRUE)
 # browseURL(naics_from_any("copper smelting", website_url=TRUE) )

  url_naics.com("copper smelting")
  ## Not run:
  naics_findwebscrape("copper smelting")
  browseURL(url_naics.com("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_validation` *Validate NAIC uploads*

Description

Validates and prepares echo uploads

Usage

```
naics_validation(NAICS_enter, NAIC_select)
```

Arguments

`NAICS`	upload validate missing and/or improper inputs

Value

boolean value (valid or not valid)

`pctile_from_raw_lookup`
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

```
pctile_from_raw_lookup(
  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 - could use data.table 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.

---

plotblocksnearby

Map view of Census blocks (their centroids) near one or more sites Utility to quickly view one or more facility points on map with the blocks found nearby

---

Description

Map view of Census blocks (their centroids) near one or more sites Utility to quickly view one or more facility points on map with the blocks found nearby

Usage

```
plotblocksnearby(sitepoints, radius = 1, usemapfast = TRUE, ...)
```

Arguments

`sitepoints`	table of points with lat, lon in decimal degrees (data.frame or data.table)
`radius`	in miles
`usemapfast`	optional. simpler plot if FALSE
`...`	passed to mapfast() or plot() depending on usemapfast

Value

invisibly returns sites2blocks like getblocksnearby() does

Examples

```
plotblocksnearby(testpoints_n(1), 2)
```

---

plot_distance_cdf_by_group

plot_distance_cdf_by_group - SLOW - needs to be optimized CDF Line Plots of cumulative share of each demographic group, within each distance Each groups distribution of distances

---

Description

plot_distance_cdf_by_group - SLOW - needs to be optimized CDF Line Plots of cumulative share of each demographic group, within each distance Each groups distribution of distances

Usage

```
plot_distance_cdf_by_group(
  results_bybg_people,
  radius_miles = round(max(x$distance_min_avgperson, na.rm = T), 1),
  demogvarname = c(namez$d, namez$d_subgroups),
  demoglabel = NULL,
  colorlist = colors()[1:length(demogvarname)],
  coloroverall = "gray"
)
```

Arguments

`results_bybg_people`	data.table from doaggregate()$results_bybg_people
`radius_miles`	miles radius that was max distance analyzed
`demogvarname`	names of columns in results_bybg_people, e.g., "pctlowinc"
`demoglabel`	friendly text names for labelling graphic, like "Low income residents"
`colorlist`	colors like "red" etc. for the demographic groups of interest
`coloroverall`	color like "gray" for everyone as a whole

Value

invisibly returns full table of sorted distances of blockgroups, cumulative count of demog groups at that block group's distance

See Also

`distance_by_groups()` `ejamit()` for examples

`plot_distance_mean_by_group`

plot_distance_mean_by_group Barplot of Average Proximity to sites, by Demographic Group (vs everyone else)

Description

Note ratio shown is ratio of distance among others to distance in group, so values below 1 mean the given demographic group lives closer to facilities.

Usage

```
plot_distance_mean_by_group(
  results_bybg_people,
  demogvarname = c(EJAM::names_d, EJAM::names_d_subgroups),
  demoglabel = NULL,
  graph = TRUE
)
```

Arguments

`results_bybg_people`	data.table from doaggregate()$results_bybg_people
`demogvarname`	vector of column names like "pctlowinc" etc.
`demoglabel`	vector of labels like "Low Income Residents" etc.

Value

data.frame with group, ratio, avg_distance_for_group, avg_distance_for_nongroup

See Also

`distance_by_group()`

`distance_by_group_plot()`

`popweightedsums` *Get population weighted sums of indicators -MOSTLY OBSOLETE The code in doaggregate() does pop wtd means faster using data.tables and* `collapse::fmean()`

Description

Get population weighted sums of indicators -MOSTLY OBSOLETE The code in doaggregate() does pop wtd means faster using data.tables and `collapse::fmean()`

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

---

`proximity.score.in.miles` *convert EJScreen proximity scores to miles per site instead of sites per kilometer Shows US percentiles if no arguments used*

Description

convert EJScreen proximity scores to miles per site instead of sites per kilometer Shows US percentiles if no arguments used

Usage

```
proximity.score.in.miles(scoresdf = NULL)
```

Arguments

`scoresdf`	data.frame of simple proximity scores like for tsdf, rmp, npl but not traffic.score or npdes one since those are weighted and not just count per km

---

`proxistat2` *Calculate a proximity score for every blockgroup Indicator of proximity of each blockgroups to some set of facilities or sites.*

Description

Calculate a proximity score for every blockgroup Indicator of proximity of each blockgroups to some set of facilities or sites.

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::

Details

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.

doaggregate() has a bit of code in it to do this same thing that proxistat2() does.

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

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` *Launch the Shiny Application in RStudio*

Description

launch Shiny web app from RStudio

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. Can be sitepoints="latlondata.xlsx" or sitepoints=testpoints_50 See `?golem::get_golem_options` for more details.

Details

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()

`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

See Also

testdata_sites2blocks testdata_sites2blocks_1000

speedtest	*speedtest Runs EJAM analysis for several radius values for various numbers of sitepoints, recording how long each step took.*

Description

speedtest Runs EJAM analysis for several radius values for various numbers of sitepoints, recording how long each step took.

Usage

```
speedtest(
  n = 10,
  sitepoints = NULL,
  weighting = "frs",
  radii = c(1, 3.106856, 5, 10, 31.06856)[1:3],
  test_getblocksnearby = TRUE,
  test_doaggregate = TRUE,
  test_batch.summarize = FALSE,
  logging = TRUE,
  logfolder = getwd(),
  logfilename = "log_n_datetime.txt",
  honk_when_ready = TRUE,
  saveoutput = FALSE,
  plot = TRUE
)
```

Arguments

`n`	optional, vector of 1 or more counts of how many random points to test, or set to 0 to interactively pick file of points in RStudio (n is ignored if sitepoints provided)
`sitepoints`	optional, (use if you do not want random points) data.frame of points or path/file with points, where columns are lat and lon in decimal degrees
`weighting`	optional, if using random points, how to weight them, such as facilities, people, or blockgroups. see `testpoints_n()`
`radii`	optional, one or more radius values in miles to use in creating circular buffers when findings residents nearby each of sitepoints. The default list includes one that is 5km (approx 3.1 miles)
`test_getblocksnearby`	whether to include this function in timing - not used because always done
`test_doaggregate`	whether to include this function in timing
`test_batch.summarize`	whether to include this function in timing
`logging`	logical optional, whether to save log file with timings of steps. NOTE this slows it down though.
`logfolder`	optional, name of folder for log file
`logfilename`	optional, name of log file to go in folder

`honk_when_ready`	optional, self-explanatory
`saveoutput`	but this slows it down if set to TRUE to save each run as .rda file
`plot`	whether to create plot of results

Value

EJAM results similar to as from the web app `ejamit()` and also creates a plot

See Also

`speedtest_plot()`

Examples

```
## Not run:
  speedseen_few <- speedtest(c(50,500), radii=c(1, 3.106856), logging=FALSE, honk=FALSE)

 speedseen_nearer_to1k <- speedtest(n = c(1e2,1e3,1e4 ), radii=c(1, 3.106856,5 ), logging=TRUE, honk=FALSE)
  save( speedseen_nearer_to1k, file = "~/../Downloads/speedseen_nearer_to1k.rda")
  rstudioapi::savePlotAsImage(        "~/../Downloads/speedseen_nearer_to1k.png")

  speedseen_all <- speedtest(
    n = c(1e2,1e3,1e4,1e5),
    radii=c(1, 3.106856, 5, 10, 31.06856),
    logging=TRUE, honk=TRUE
  )

## End(Not run)
```

---

speedtest_plot   *utility to plot output of speedtest(), rate of points analyzed per hour*

Description

utility to plot output of speedtest(), rate of points analyzed per hour

Usage

```
speedtest_plot(x, ltype = "b")
```

Arguments

`x`	table from speedtest()
`ltype`	optional type of line for plot

Value

side effect is a plot. returns x but with seconds column added to it

See Also

`speedtest()`

`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.frame of 100 percentiles and means for each US State and PR and DC.*

Description

data.frame 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, except it also has data for the demographic race/ethnicity subgroups. For details on how the table was made, see /EJAM/data-raw/usastats_subgroups.R

`statestats_query` *convenient way to see mean, pctiles of Env or Demog indicators from lookup table*

Description

convenient way to see mean, pctiles of Env or Demog indicators from lookup table

Usage

```
statestats_query(
  ST = sort(unique(EJAM::statestats$REGION)),
  varnames = c(EJAM::names_e, EJAM::names_d),
  PCTILES = NULL,
  dig = 2
)
```

Arguments

`ST`	vector of state abbreviations, or USA
`varnames`	names of columns in lookup table, like "proximity.rmp"
`PCTILES`	vector of percentiles 0-100 and/or "mean"
`dig`	digits to round to

Examples

```
## Not run:
## in USA overall, see mean and key percentiles for all demog and envt indicators
usastats_query() # or statestats_query('us') # can say us or US or USA or usa etc.
usastats_query(PCTILES = 'mean')
usastats_means() # same but nicer looking format in console
usastats_means(dig=4)
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_d)]
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_e)]
## in 1 state, see mean and key percentiles for all demog and envt indicators
statestats_query('MD')
```

```
## in 1 state, see mean and key percentiles for just demog indicators
statestats_queryd('MD')
## 1 indicator in 1 state, see a few key percentiles and mean
statestats_query('MD','proximity.tsdf')
## mean of 1 indicator for each state
statestats_query(varnames = 'proximity.tsdf')
## using full blockgroup dataset, not lookup tables of percentiles,
blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)),  .SDcols= c(names_d, names_e)]
##   see all total counts (not just US means),
##   demographics including subgroups,
##   but not environmental indicators.
t(round(EJAMbatch.summarizer::ustotals2(bg = EJAM::blockgroupstats),2))
t(blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)),  .SDcols= c(names_e, names_d)])


## End(Not run)
```

`statestats_queryd` *convenient way to see mean, pctiles of DEMOG indicators from lookup table*

Description

convenient way to see mean, pctiles of DEMOG indicators from lookup table

Usage

```
statestats_queryd(
  ST = sort(unique(EJAM::statestats$REGION)),
  varnames = EJAM::names_d,
  PCTILES = NULL,
  dig = 2
)
```

Arguments

ST	vector of state abbreviations, or USA
varnames	names of columns in lookup table, like "proximity.rmp"
PCTILES	vector of percentiles 0-100 and/or "mean"
dig	digits to round to

Examples

```
## Not run:
## in USA overall, see mean and key percentiles for all demog and envt indicators
usastats_query() # or statestats_query('us') # can say us or US or USA or usa etc.
usastats_query(PCTILES = 'mean')
usastats_means() # same but nicer looking format in console
usastats_means(dig=4)
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_d)]
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_e)]
## in 1 state, see mean and key percentiles for all demog and envt indicators
```

```
statestats_query('MD')
## in 1 state, see mean and key percentiles for just demog indicators
statestats_queryd('MD')
## 1 indicator in 1 state, see a few key percentiles and mean
statestats_query('MD','proximity.tsdf')
## mean of 1 indicator for each state
statestats_query(varnames = 'proximity.tsdf')
## using full blockgroup dataset, not lookup tables of percentiles,
blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)),  .SDcols= c(names_d, names_e)]
##   see all total counts (not just US means),
##   demographics including subgroups,
##   but not environmental indicators.
t(round(EJAMbatch.summarizer::ustotals2(bg = EJAM::blockgroupstats),2))
t(blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)),  .SDcols= c(names_e, names_d)])


## End(Not run)
```

---

statestats_querye — *convenient way to see mean, pctiles of ENVIRONMENTAL indicators from lookup table*

Description

convenient way to see mean, pctiles of ENVIRONMENTAL indicators from lookup table

Usage

```
statestats_querye(
  ST = sort(unique(EJAM::statestats$REGION)),
  varnames = EJAM::names_e,
  PCTILES = NULL,
  dig = 2
)
```

Examples

```
## Not run:
## in USA overall, see mean and key percentiles for all demog and envt indicators
usastats_query() # or statestats_query('us') # can say us or US or USA or usa etc.
usastats_query(PCTILES = 'mean')
usastats_means() # same but nicer looking format in console
usastats_means(dig=4)
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_d)]
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_e)]
## in 1 state, see mean and key percentiles for all demog and envt indicators
statestats_query('MD')
## in 1 state, see mean and key percentiles for just demog indicators
statestats_queryd('MD')
## 1 indicator in 1 state, see a few key percentiles and mean
statestats_query('MD','proximity.tsdf')
## mean of 1 indicator for each state
statestats_query(varnames = 'proximity.tsdf')
```

```
## using full blockgroup dataset, not lookup tables of percentiles,
blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)),  .SDcols= c(names_d, names_e)]
##   see all total counts (not just US means),
##   demographics including subgroups,
##   but not environmental indicators.
t(round(EJAMbatch.summarizer::ustotals2(bg = EJAM::blockgroupstats),2))
t(blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)),  .SDcols= c(names_e, names_d)])


## End(Not run)
```

`states_infer` *states_infer Get cleaned table of US State etc. by siteid, from lat/lon, or from FIPS*

Description

states_infer Get cleaned table of US State etc. by siteid, from lat/lon, or from FIPS

Usage

```
states_infer(x)
```

Arguments

x	data.frame or data.table with either ST column or lat and lon columns, or FIPS, and optionally a column with siteid or column called n

Value

data.frame with unique siteid, ST, etc.

See Also

`state_from_latlon()` `state_from_fips()`

`states_shapefile` *US States boundaries 2020 shapefile from TIGER*

Description

This is used to figure out which state contains each point (facility/site).

Details

This is used by `state_from_latlon()` to find which state is associated with each point that the user wants to analyze. That is needed to report indicators in the form of State-specific percentiles (e.g., a score that is at the 80th percentile within Texas). It is created by the package via EJAM/data-raw/make_states_shapefile.R Created roughly as follows:

setwd("~/../../R/mysource/EJAM/data-raw") download.file("https://www.census.gov/cgi-bin/geo/shapefiles/index.php?y
destfile = "tl_2020_us_state.zip") dir.create("./shp") unzip("tl_2020_us_state.zip", exdir = "./shp")
states_shapefile <- sf::st_read("./shp", quiet = FALSE) class(states_shapefile)

See Also

`state_from_latlon()` `get_blockpoints_in_shape()`

`state_from_blockid` *state_from_blockid given vector of blockids, get state abbreviation of each*

Description

state_from_blockid given vector of blockids, get state abbreviation of each

Usage

```
state_from_blockid(blockid)
```

Arguments

`blockid`	vector of blockid values as from EJAM::blockpoints

Value

vector of ST info like AK, CA, DE, etc.

Examples

```
state_from_blockid(c(8174952, blockpoints$blockid[5:6]))
```

`state_from_blocktable` *state_from_blocktable given data.table with blockid column, get state abbreviation of each*

Description

state_from_blocktable given data.table with blockid column, get state abbreviation of each

Usage

```
state_from_blocktable(dt_with_blockid)
```

Arguments

`dt_with_blockid`	

Value

vector of ST info like AK, CA, DE, etc.

Examples

```
state_from_blocktable(blockpoints[45:49,])
```

`state_from_fips` *state_from_fips*

Description

state_from_fips

Usage

```
state_from_fips(fips)
```

Arguments

`fips`	Census FIPS codes vector, numeric or char, 2-digit, 5-digit, etc. OK

Value

vector of 2-character state abbreviations like CA,MD,TX,OH

`state_from_latlon` *find what state is where each point is located Takes 3 seconds to find state for 1k points, so a faster alternative would be useful*

Description

find what state is where each point is located Takes 3 seconds to find state for 1k points, so a faster alternative would be useful

Usage

```
state_from_latlon(lat, lon, states_shapefile = EJAM::states_shapefile)
```

Arguments

`lat`	latitudes vector
`lon`	longitudes vector
`shapefile`	shapefile of US States, in package already

Value

Returns data.frame: ST, statename, FIPS.ST, REGION, n as many rows as elements in lat or lon

See Also

states_shapefile `get_blockpoints_in_shape()` `states_infer()`

`ST_by_site_from_sites2blocks`
Get State that each site is in, from a table of siteid, blockid, distance

Description

Find the 2-character State abbreviation for each site. This is for when you need to know the state each site is in, to be able to report state percentiles, but you do not have the original list of siteid lat/lon or State info. This can infer the State each site is located in, based on the state of the nearest block (and its parent blockgroup).

Usage

```
ST_by_site_from_sites2blocks(sites2blocks)
```

Arguments

`sites2blocks`	data.table or data.frame, like sites2blocks_example, from `getblocksnearby()` that has columns siteid and blockid and distance

Value

data.table with columns siteid, ST

Examples

```
  ST_by_site_from_sites2blocks(sites2blocks_example)
  ## Not run:
 fname = './inst/testdata/testpoints_207_sites_with_signif_violations_NAICS_326_ECHO.csv'
  x = ST_by_site_from_sites2blocks(
    getblocksnearby( latlon_from_anything(fname), quadtree = localtree))
  y = read_csv_or_xl(fname)
  x$ST == y$FacState

## End(Not run)
```

`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

`testdata_sites2blocks` *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

See Also

testdata_sites2blocks testdata_sites2blocks_1000

`testdata_sites2blocks_1000`
data.table of output of getblocknearby(testpoints_1000,cutoff = 3.1), each row is a unique site-block-distance

---

Description

data.table of output of getblocknearby(testpoints_1000,cutoff = 3.1), each row is a unique site-block-distance

See Also

testdata_sites2blocks

`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_n`	*Random points on the map Get data.table of Random Points (lat lon) for Testing/ Benchmarking/ Demos, weighted in various ways. The weighting can be specified so that each point reflects the average EPA-regulated facility, blockgroup, block, place on the map, or US resident.*

Description

Random points on the map Get data.table of Random Points (lat lon) for Testing/ Benchmarking/ Demos, weighted in various ways. The weighting can be specified so that each point reflects the average EPA-regulated facility, blockgroup, block, place on the map, or US resident.

Usage

```
testpoints_n(
  n = 10,
  weighting = c("frs", "pop", "area", "bg", "block"),
  dt = TRUE,
  ST_of_blockgroup = NULL
)
```

Arguments

`n` Number of points needed (sample size)

`weighting` word indicating how to weight the random points (some synonyms are allowed, in addition to those shown here):

Note the default is frs, but you may want to use pop even though it is slower.

- pop or people = Average Person: random person among all US residents (block point of residence per 2020 Census)
- frs or facility = Average Facility: random EPA-regulated facility from actives in Facility Registry Services (FRS)
- bg = Average Blockgroup: random US Census block group (internal point like a centroid)
- block = Average Block: random US Census block (internal point like a centroid)
- area or place = Average Place: random point on a map (internal point of avg blockgroup weighted by its square meters size)

`dt` logical, whether to return a data.table (DEFAULT) instead of normal data.frame

`ST_of_blockgroup`
optional, can be a character vector of 2 letter State abbreviations to pick from only some States.

Value

data.frame or data.table with columns lat, lon in decimal degrees, and any other columns that are in the table used (based on weighting)

Examples

```
## Not run:
mapfast(testpoints_n(300, ST_of_blockgroup = c('LA','MS')) )
n=2
for (d in c(TRUE,FALSE)) {
  for (w in c('frs', 'pop', 'area', 'bg', 'block')) {
    cat("n=",n,"  weighting=",w, "  dt=",d,"\n\n")
    print(x <- testpoints_n(n,w,d)); print(class(x))
    cat('\n')
  }
}

## End(Not run)
```

`test_regid` *test data, vector of EPA FRS Registry ID numbers*

Description

test data, vector of EPA FRS Registry ID numbers

Details

For testing, e.g.,

frs_from_siteid(test_regid)

mapfast(frs_from_regid(test_regid))

`url_bookmark_save` *url_bookmark_save save bookmarked EJScreen session (map location and indicator)*

Description

url_bookmark_save save bookmarked EJScreen session (map location and indicator)

Usage

```
url_bookmark_save(..., file = "ejscreenbookmark.json")
```

Arguments

`...`	passed to `url_bookmark_text()`
`file`	path and name of .json file you want to save locally

Details

WORK IN PROGRESS - NOT USED AS OF EARLY 2023. You can use this function to create and save a json file that is a bookmark for a specific place/ map view/ data layer in EJScreen. You can later pull up that exact map in EJScreen by launching EJScreen, clicking Tools, Save Session, Load from File.

***Units are not lat lon: "spatialReference":"latestWkid":3857,"wkid":102100

Note: (1) The number of sessions that can be saved depends on the browser cache size. (2) Session files, if saved, are available from the default Downloads folder on your computer. (3) Users should exercise caution when saving sessions that may contain sensitive or confidential data.

Value

URL for 1 bookmarked EJScreen map location and variable displayed on map

---

`url_bookmark_text` *url_bookmark_text URL for 1 bookmarked EJScreen session (map location and indicator)*

Description

url_bookmark_text URL for 1 bookmarked EJScreen session (map location and indicator)

Usage

```
url_bookmark_text(
  x = c(-13232599.1784247, -13085305.0249191),
  y = c(3970069.24597194, 4067373.582979),
  name = "BookmarkedEJScreenMap",
  title = "Socioeconomic Indicators",
  renderField = "B_UNEMPPCT",
  pctlevel = "nation",
  xmin = 1.1 * min(x),
  xmax = 0.9 * min(x),
  ymin = 0.9 * min(y),
  ymax = 1.1 * min(y),
  urlrest =
   "https://geopub.epa.gov/arcgis/rest/services/ejscreen/ejscreen_v2022_with_AS_CNMI_GU_VI/MapSe
)
```

Arguments

`x`	vector of approx topleft, bottomright longitudes in some units EJScreen uses? Units are not lat lon: "spatialReference":"latestWkid":3857,"wkid":102100
`y`	vector of approx topleft, bottomright latitudes in some units EJScreen uses? Units are not lat lon: "spatialReference":"latestWkid":3857,"wkid":102100
`name`	Your name for the map bookmark
`title`	Your name for the map like Socioeconomic Indicators or Pollution and Sources
`renderField`	name of variable shown on map, like B_UNEMPPCT for map color bins of percent unemployed or B_PTRAF for traffic indicator

`pctlevel`	nation or state
`xmin`	calculated bounding box for map view
`xmax`	calculated bounding box for map view
`ymin`	calculated bounding box for map view
`ymax`	calculated bounding box for map view
`urlrest`	Just use the default but it changes each year

Details

WORK IN PROGRESS - NOT USED AS OF EARLY 2023. You can use this function to create and save a json file that is a bookmark for a specific place/ map view/ data layer in EJScreen. You can later pull up that exact map in EJScreen by launching EJScreen, clicking Tools, Save Session, Load from File.

Note: (1) The number of sessions that can be saved depends on the browser cache size. (2) Session files, if saved, are available from the default Downloads folder on your computer. (3) Users should exercise caution when saving sessions that may contain sensitive or confidential data.

Value

URL for 1 bookmarked EJScreen map location and variable displayed on map

See Also

`url_bookmark_save()`

Examples

```
## Not run:
  url_bookmark_text()
  url_bookmark_save(
    x=c(-10173158.179197036, -10128824.702791695),
    y=c(3548990.034736070,3579297.316451102),
    file="./mysavedejscreensession1.json")

## End(Not run)
```

`url_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_Map_Service_Layer/02ss0000000r000000/`

Usage

```
url_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 url_getacs_epaquery_chunked()

Value

table

Examples

```
url_getacs_epaquery(justurl=TRUE)
```

---

url_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

```
url_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 url_getacs_epaquery()

Value

table

Examples

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

`url_get_eparest_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

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

Arguments

`objectIds`	see API
`chunksize`	see API
`...`	passed to url_getacs_epaquery()

Value

a table

`url_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

```
url_get_via_url(url)
```

Arguments

`url`	the url for an EJScreen ejquery request

Value

json

`url_naics.com` *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

```
url_naics.com(query, as_html = FALSE, linktext)
```

Arguments

`query`	string query term like "gasoline" or "copper smelting"
`as_html`	Whether to return as just the urls or as html hyperlinks to use in a DT::datatable() for example
`linktext`	used as text for hyperlinks, if supplied and as_html=TRUE

Value

URL as string

See Also

`url_linkify()`

usastats *data.frame of 100 percentiles and means*

Description

data.frame 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, except it also has data for the demographic race/ethnicity subgroups. For details on how the table was made, see /EJAM/data-raw/usastats_subgroups.R

usastats_means *convenient way to see USA MEANS of ENVIRONMENTAL and DEMOGRAPHIC indicators from lookup table*

Description

convenient way to see USA MEANS of ENVIRONMENTAL and DEMOGRAPHIC indicators from lookup table

Usage

```
usastats_means(...)
```

Arguments

`...`	Arguments passed on to `usastats_query` `varnames` names of columns in lookup table, like "proximity.rmp" `PCTILES` vector of percentiles 0-100 and/or "mean"

usastats_query *convenient way to see USA mean, pctiles of Env and Demog indicators from lookup table*

Description

convenient way to see USA mean, pctiles of Env and Demog indicators from lookup table

Usage

```
usastats_query(
  varnames = c(EJAM::names_e, EJAM::names_d),
  PCTILES = NULL,
  dig = 2
)
```

Arguments

`varnames`	names of columns in lookup table, like "proximity.rmp"
`PCTILES`	vector of percentiles 0-100 and/or "mean"
`@dig`	how many digits to round to

Examples

```
## Not run:
## in USA overall, see mean and key percentiles for all demog and envt indicators
usastats_query() # or statestats_query('us') # can say us or US or USA or usa etc.
usastats_query(PCTILES = 'mean')
usastats_means() # same but nicer looking format in console
usastats_means(dig=4)
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_d)]
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_e)]
## in 1 state, see mean and key percentiles for all demog and envt indicators
statestats_query('MD')
## in 1 state, see mean and key percentiles for just demog indicators
statestats_queryd('MD')
## 1 indicator in 1 state, see a few key percentiles and mean
statestats_query('MD','proximity.tsdf')
## mean of 1 indicator for each state
statestats_query(varnames = 'proximity.tsdf')
## using full blockgroup dataset, not lookup tables of percentiles,
blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)), .SDcols= c(names_d, names_e)]
##   see all total counts (not just US means),
##   demographics including subgroups,
##   but not environmental indicators.
t(round(EJAMbatch.summarizer::ustotals2(bg = EJAM::blockgroupstats),2))
t(blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)), .SDcols= c(names_e, names_d)])


## End(Not run)
```

---

usastats_queryd *convenient way to see USA mean, pctiles of DEMOGRAPHIC indicators from lookup table*

Description

convenient way to see USA mean, pctiles of DEMOGRAPHIC indicators from lookup table

Usage

```
usastats_queryd(varnames = EJAM::names_d, PCTILES = NULL, dig = 2)
```

Arguments

`varnames`	names of columns in lookup table, like "proximity.rmp"
`PCTILES`	vector of percentiles 0-100 and/or "mean"
`@dig`	how many digits to round to

Examples

```
## Not run:
## in USA overall, see mean and key percentiles for all demog and envt indicators
usastats_query() # or statestats_query('us') # can say us or US or USA or usa etc.
usastats_query(PCTILES = 'mean')
usastats_means() # same but nicer looking format in console
usastats_means(dig=4)
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_d)]
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_e)]
## in 1 state, see mean and key percentiles for all demog and envt indicators
statestats_query('MD')
## in 1 state, see mean and key percentiles for just demog indicators
statestats_queryd('MD')
## 1 indicator in 1 state, see a few key percentiles and mean
statestats_query('MD','proximity.tsdf')
## mean of 1 indicator for each state
statestats_query(varnames = 'proximity.tsdf')
## using full blockgroup dataset, not lookup tables of percentiles,
blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)),  .SDcols= c(names_d, names_e)]
##   see all total counts (not just US means),
##   demographics including subgroups,
##   but not environmental indicators.
t(round(EJAMbatch.summarizer::ustotals2(bg = EJAM::blockgroupstats),2))
t(blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)),  .SDcols= c(names_e, names_d)])


## End(Not run)
```

`usastats_querye` *convenient way to see USA mean, pctiles of ENVIRONMENTAL indicators from lookup table*

Description

convenient way to see USA mean, pctiles of ENVIRONMENTAL indicators from lookup table

Usage

```
usastats_querye(varnames = EJAM::names_e, PCTILES = NULL, dig = 2)
```

Arguments

`varnames`	names of columns in lookup table, like "proximity.rmp"
`PCTILES`	vector of percentiles 0-100 and/or "mean"
`@dig`	how many digits to round to

Examples

```
## Not run:
## in USA overall, see mean and key percentiles for all demog and envt indicators
usastats_query() # or statestats_query('us') # can say us or US or USA or usa etc.
usastats_query(PCTILES = 'mean')
```

```
usastats_means() # same but nicer looking format in console
usastats_means(dig=4)
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_d)]
usastats[!(usastats$PCTILE < 50), c("PCTILE", names_e)]
## in 1 state, see mean and key percentiles for all demog and envt indicators
statestats_query('MD')
## in 1 state, see mean and key percentiles for just demog indicators
statestats_queryd('MD')
## 1 indicator in 1 state, see a few key percentiles and mean
statestats_query('MD','proximity.tsdf')
## mean of 1 indicator for each state
statestats_query(varnames = 'proximity.tsdf')
## using full blockgroup dataset, not lookup tables of percentiles,
blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)),  .SDcols= c(names_d, names_e)]
##   see all total counts (not just US means),
##   demographics including subgroups,
##   but not environmental indicators.
t(round(EJAMbatch.summarizer::ustotals2(bg = EJAM::blockgroupstats),2))
t(blockgroupstats[,  lapply(.SD, function(x) mean(x, na.rm=T)),  .SDcols= c(names_e, names_d)])


## End(Not run)
```

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

`xls_formatting` *Format batch results for excel also see other functions related to this!*

Description

Format batch results for excel also see other functions related to this!

Usage

```
xls_formatting(
  df,
  hyperlink_cols = NULL,
  heatmap_colnames = NULL,
  heatmap_cuts = c(80, 90, 95),
  heatmap_colors = c("yellow", "orange", "red")
)
```

Arguments

`df`	data.frame, table of batch buffer results
`hyperlink_cols`	vector of names of columns in df to get treated as hyperlinks in excel
`heatmap_colnames`	vector of names of columns in df to apply conditional formatting to, by coloring like a heatmap.
`heatmap_cuts`	vector of color names for heatmap bins, same length as heatmap_cuts, where first color is for those >= 1st cutpoint, but <2d, second color is for those >=2d cutpoint but <3d, etc.
`heatmap_colors`	vector of colors corresponding to cuts

Value

A workbook via openxlsx::writeData() ready to be saved via openxlsx::saveWorkbook()

`xls_formatting2` *xls_formatting2 Format EJAM tabular outputs for saving as Excel spreadsheet*

Description

xls_formatting2 Format EJAM tabular outputs for saving as Excel spreadsheet

Usage

```
xls_formatting2(
  overall,
  eachsite,
  graycolnums = NULL,
  narrowcolnums = NULL,
  graycolor = "gray",
  narrow6 = 6,
  ...
)
```

Arguments

`overall`	table to save in one tab, from EJAM analysis of indicators overall (one row)
`eachsite`	table to save in another tab, from EJAM analysis site by site (one row per site)
`graycolnums`	which columns to deemphasize
`narrowcolnums`	which columns to make narrow
`graycolor`	color used to deemphasize some columns
`narrow6`	how narrow
`...`	other params passed along to `openxlsx::writeData()`

Details

NEED TO MERGE THIS WITH EJAMejscreenapi::xls_formatting_api()

Value

a workbook, ready to be saved in spreadsheet format, with tabs like "Overall" and "Each Site"

Examples

```
## Not run:
 wb <- xls_formatting2(datasetResults()$results_overall, datasetResults()$results_bysite)
  openxlsx::saveWorkbook(wb, "results.xlsx", overwrite = TRUE)

## End(Not run)
```

Index