

Package ‘PWFSLSmoke’

July 19, 2019

Type Package

Version 1.2.100

Title Utilities for Working with Air Quality Monitoring Data

Author Jonathan Callahan [aut, cre],
Rohan Aras [aut],
Zach Dingels [aut],
Jon Hagg [aut],
Jimin Kim [aut],
Hans Martin [aut],
Helen Miller [aut],
Spencer Pease [aut],
Rex Thompson [aut],
Alice Yang [aut]

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

Depends R (>= 3.3.0), dplyr, maps, MazamaCoreUtils (>= 0.1.3),
MazamaSpatialUtils (>= 0.6.1)

Imports curl, cluster, dygraphs (>= 1.1.1.4), geosphere, ggmap, glue,
httr, jsonlite, leaflet (>= 1.0.0), lubridate, magrittr,
mapproj, mapproj, methods, openair, png, purrr, raster,
RColorBrewer, rgdal, RgoogleMaps, readr, reshape2, rlang, sf,
slippymath, sp, stringr, tibble, tidyr, xts

Suggests knitr, lintr, rmarkdown, testthat

Description Utilities for working with air quality monitoring data with a focus on small particulates (PM2.5) generated by wildfire smoke. Functions are provided for downloading available data from the United States 'EPA' <<https://www.epa.gov/outdoor-air-quality-data>> and it's 'AirNow' air quality site <<https://www.airnow.gov>>. Additional sources of PM2.5 data made accessible by the package include: 'AIRSIS' (password protected) <<https://www.oceanering.com/data-management/>> and 'WRCC' <<https://wrcc.dri.edu/cgi-bin/smoke.pl>>. Data compilations are provided by 'PWFSL' <<https://www.fs.fed.us/pnw/pwfs/>>.

License GPL-3

VignetteBuilder knitr

Repository CRAN

URL <https://github.com/MazamaScience/PWFSLSmoke>

BugReports <https://github.com/MazamaScience/PWFSLSmoke/issues>

Encoding UTF-8

LazyData true

RoxygenNote 6.1.1

NeedsCompilation no

Date/Publication 2019-07-18 23:10:02 UTC

R topics documented:

addAQILegend	2
addAQILines	3
addAQIStackedBar	3
addBullseye	4
addIcon	4
addMarker	5
addPolygon	6
addShadedBackground	7
addShadedNight	7
addWindBarbs	8
airnow_createDataDataframes	9
airnow_createMetaDataframes	10
airnow_createMonitorObjects	12
airnow_downloadHourlyData	14
airnow_downloadParseData	15
airnow_downloadSites	16
airnow_load	17
airnow_loadAnnual	18
airnow_loadDaily	19
airnow_loadLatest	20
airnow_qualityControl	21
AIRSIS	22
airsis_availableUnits	23
airsis_BAM1020QualityControl	24
airsis_createDataDataframe	25
airsis_createMetaDataframe	25
airsis_createMonitorObject	26
airsis_createRawDataframe	28
airsis_downloadData	30
airsis_EBAMQualityControl	31
airsis_ESAMQualityControl	32
airsis_identifyMonitorType	33

airsis_load	34
airsis_loadAnnual	34
airsis_loadDaily	35
airsis_loadLatest	36
airsis_parseData	37
airsis_qualityControl	38
AQI	39
aqiColors	40
aqiPalette	41
AQI_en	42
AQI_es	42
Carmel_Valley	43
CONUS	43
distance	44
epa_createDataDataframe	45
epa_createMetaDataframe	45
epa_createMonitorObject	46
epa_downloadData	47
epa_load	48
epa_loadAnnual	49
epa_parseData	50
esriToken	52
generic_downloadData	52
generic_parseData	53
getEsriToken	54
getGoogleApiKey	55
googleApiKey	55
initializeMazamaSpatialUtils	56
loadDaily	56
loadLatest	57
monitor_aqi	58
monitor_asDataframe	59
monitor_collapse	60
monitor_combine	61
monitor_dailyBarplot	62
monitor_dailyStatistic	63
monitor_dailyStatisticList	64
monitor_dailyThreshold	66
monitor_distance	67
monitor_downloadAnnual	68
monitor_downloadDaily	69
monitor_downloadLatest	70
monitor_dygraph	71
monitor_extractDataFrame	72
monitor_getCurrentStatus	73
monitor_getDailyMean	75
monitor_hourlyBarplot	75
monitor_isEmpty	77

monitor_isMonitor	77
monitor_isolate	78
monitor_isTidy	79
monitor_join	80
monitor_leaflet	81
monitor_load	82
monitor_loadAnnual	83
monitor_loadDaily	84
monitor_loadLatest	86
monitor_map	87
monitor_nowcast	88
monitor_performance	90
monitor_performanceMap	91
monitor_print	92
monitor_reorder	93
monitor_replaceData	94
monitor_rollingMean	94
monitor_rollingMeanPlot	95
monitor_scaleData	97
monitor_stamenmap	97
monitor_staticmap	99
monitor_subset	100
monitor_subsetBy	101
monitor_subsetByDistance	102
monitor_subsetData	103
monitor_subsetMeta	104
monitor_timeAverage	104
monitor_timeInfo	105
monitor_timeseriesPlot	106
monitor_toTidy	108
monitor_trim	109
monitor_writeCSV	109
monitor_writeCurrentStatusGeoJSON	110
Northwest_Megafires	111
parseDatetime	112
rawPlot_pollutionRose	113
rawPlot_timeOfDaySpaghetti	114
rawPlot_timeseries	115
rawPlot_windRose	116
raw_enhance	117
raw_getHighlightDates	118
setEsriToken	118
setGoogleApiKey	119
skill_confusionMatrix	120
skill_ROC	121
skill_ROCPlot	123
staticmap_getEsrimapBrick	124
staticmap_getRasterBrick	125

staticmap_getStamenmapBrick	127
staticmap_plotRasterBrick	129
tidy_toMonitor	130
timeInfo	130
upgradeMeta_v1.0	132
US_52	132
WRCC	133
wrcc_createDataDataframe	133
wrcc_createMetaDataframe	134
wrcc_createMonitorObject	135
wrcc_createRawDataframe	137
wrcc_downloadData	138
wrcc_EBAMQualityControl	139
wrcc_ESAMQualityControl	140
wrcc_identifyMonitorType	141
wrcc_load	142
wrcc_loadAnnual	143
wrcc_loadDaily	144
wrcc_loadLatest	145
wrcc_parseData	146
wrcc_qualityControl	147

addAQILegend	<i>Add an AQI Legend to a Map</i>
--------------	-----------------------------------

Description

This function is a convenience wrapper around `graphics::legend()`. It will show the AQI colors and names by default if `col` and `legend` are not specified.

Usage

```
addAQILegend(x = "topright", y = NULL, col = rev(AQI$colors),
             legend = rev(AQI$names), pch = 16, title = "Air Quality Index",
             ...)
```

Arguments

<code>x</code>	<code>x</code> coordinate passed on to the <code>legend()</code> command
<code>y</code>	<code>y</code> coordinate passed on to the <code>legend()</code> command
<code>col</code>	the color for points/lines in the legend
<code>legend</code>	a character vector to be shown in the legend
<code>pch</code>	plotting symbols in the legend
<code>title</code>	title for the legend
<code>...</code>	additional arguments to be passed to <code>legend()</code>

addAQIILines *Add AQI Lines to a Plot*

Description

This function is a convenience for:

```
abline(h = AQI$breaks_24, col = AQI$colors)
```

Usage

```
addAQIILines(...)
```

Arguments

... additional arguments to be passed to `abline()`

addAQIStackedBar *Create Stacked AQI Bar*

Description

Draws a stacked bar indicating AQI levels on one side of a plot

Usage

```
addAQIStackedBar(width = 0.01, height = 1, pos = "left")
```

Arguments

width width of the bar as a fraction of the width of the plot area (default = .02)
height height of the bar as a fraction of the height of the plot area (default = 1)
pos position of the stacked bar. Either 'left' or 'right'

Value

Stacked AQI Bar

addBullseye *Add a Bullseyes to a Map or RgoogleMap Plot*

Description

Draws a bullseye with concentric circles of black and white.

Usage

```
addBullseye(longitude, latitude, map = NULL, cex = 2, lwd = 2)
```

Arguments

longitude	vector of longitudes
latitude	vector of latitudes
map	optional RgoogleMaps map object
cex	character expansion
lwd	line width of individual circles

Examples

```
wa <- monitor_subset(Northwest_Megafires, stateCodes='WA', tlim=c(20150821,20150828))
monitor_map(wa, cex=4)
addBullseye(wa$meta$longitude, wa$meta$latitude)
```

addIcon *Add Icons to a Map or RgoogleMap Plot*

Description

Adds an icon to map – an RgoogleMaps map object. The following icons are available:

- orangeFlame – yellow-orange flame
- redFlame – orange-red flame

You can use other .png files as icons by passing an absolute path as the icon argument.

Usage

```
addIcon(icon, longitude, latitude, map = NULL, expansion = 0.1,
        pos = 0)
```

Arguments

icon	object to be plotted
longitude	vector of longitudes
latitude	vector of latitudes
map	optional RgoogleMaps map object
expansion	icon expansion factor
pos	position of icon relative to location (0=center, 1=bottom, 2=left, 3=top,4=right)

Note

For RgoogleMaps, the `expansion` will be ~ 0.1 while for basic plots it may need to be much smaller, perhaps ~ 0.001.

Examples

```
## Not run:
ca <- loadLatest() %>% monitor_subset(stateCodes='CA')
# Google map
map <- monitor_esriMap(ca)
addIcon("orangeFlame", ca$meta$longitude, ca$meta$latitude, map=map, expansion=0.1)
# line map
monitor_map(ca)
addIcon("orangeFlame", ca$meta$longitude, ca$meta$latitude, expansion=0.002)

## End(Not run)
```

addMarker

Add Icons to a Map or RgoogleMap Plot

Description

Adds a marker to a plot or map – an RgoogleMaps map object or Raster* object.

Usage

```
addMarker(longitude, latitude, color = "red", map = NULL,
          expansion = 1, ...)
```

Arguments

longitude	vector of longitudes
latitude	vector of latitudes
color	marker color: 'red', 'green', 'yellow', 'orange', or 'blue'. Also includes AQI category colors, specified 'AQI[number]' eg. 'AQI1'
map	optional RgoogleMaps map object or Raster* object
expansion	icon expansion factor. Ignored if width and height are specified.
...	arguments passed on to rasterImage

Examples

```
## Not run:
ca <- loadLatest() %>% monitor_subset(stateCodes='CA')
# Google map
map <- monitor_esriMap(ca)
addMarker(ca$meta$longitude, ca$meta$latitude, map=map)
# line map
monitor_map(ca)
addMarker(ca$meta$longitude, ca$meta$latitude, color = "blue", expansion = 1)

## End(Not run)
```

addPolygon

Add a Colored Polygon to a Plot

Description

Add a multi-sided polygon to a plot.

Usage

```
addPolygon(x = 0, y = 0, sides = 72, radius = 1, rotation = 0,
           border = NULL, col = NA, ...)
```

Arguments

x	x location of center
y	y location of center
sides	number of sides
radius	radius
rotation	amount to rotate the polygon in radians
border	border color (see ?polygon)
col	fill color (see ?polygon)
...	additional arguments to be passed to polygon()

Examples

```
# Create AQI dots
plot(1:6, rep(0,6), xlim=c(-1,7), ylim=c(-1,3),
     axes=FALSE, xlab='', ylab='', col='transparent')
for (i in 1:6) {
  addPolygon(i, 2, 72, 0.4, 0, col=PWFSLSmoke::AQI$colors[i])
  addPolygon(i, 1, 4, 0.4, pi/4, co=PWFSLSmoke::AQI$colors[i])
  addPolygon(i, 0, 3, 0.4, pi/2, col=PWFSLSmoke::AQI$colors[i])
}
```

 addShadedBackground

Add Shaded Background to a Plot

Description

Adds vertical lines to an existing plot using any variable that shares the same length as the time axis of the current plot. Line widths corresponds to magnitude of values.

Usage

```
addShadedBackground(param, timeAxis, breaks = stats::quantile(param,
  na.rm = TRUE), col = "blue", maxOpacity = 0.2, lwd = 1)
```

Arguments

param	vector of data to be represented
timeAxis	vector of times of the same length as param
breaks	set of breaks used to assign colors
col	color for vertical lines
maxOpacity	maximum opacity
lwd	line width

 addShadedNight

Add Nighttime Shading to a Plot

Description

Draw shading rectangles on a plot to indicate nighttime hours.

Usage

```
addShadedNight(timeInfo, col = adjustcolor("black", 0.1))
```

Arguments

timeInfo	dataframe with local time, sunrise, and sunset
col	color used to shade nights – defaults to <code>adjustcolor('black', 0.2)</code>

See Also

timeInfo

addWindBarbs *Add wind barbs to a map*

Description

Add a multi-sided polygon to a plot.

Usage

```
addWindBarbs(x, y, speed, dir, circleSize = 1,
             circleFill = "transparent", lineCol = 1, extraBarbLength = 0,
             barbSize = 1, ...)
```

Arguments

x	vector of longitudes
y	vector of latitudes
speed	vector of wind speeds in knots
dir	wind directions in degrees clockwise from north
circleSize	size of the circle
circleFill	circle fill color
lineCol	line color (currently not supported)
extraBarbLength	add length to barbs
barbSize	size of the barb
...	additional arguments to be passed to lines

References

https://commons.wikimedia.org/wiki/Wind_speed

Examples

```
maps::map('state', "washington")
x <- c(-121, -122)
y <- c(47.676057, 47)
addWindBarbs(x, y, speed = c(45,65), dir = c(45, 67),
             circleSize = 1.8, circleFill = c('orange', 'blue'))
```

`airnow_createDataDataframes`*Return reshaped dataframes of AirNow data*

Description

This function uses the `airnow_downloadParseData` function to download monthly dataframes of AirNow data and restructures that data into a format that is compatible with the `PWFSLSmoke` package `ws_monitor` data model.

AirNow data parameters include at least the following list:

1. BARPR
2. BC
3. CO
4. NO
5. NO2
6. NO2Y
7. NO2X
8. NOX
9. NOOY
10. OC
11. OZONE
12. PM10
13. PM2.5
14. PRECIP
15. RHUM
16. SO2
17. SRAD
18. TEMP
19. UV-AETH
20. WD
21. WS

Setting `parameters=NULL` will generate a separate dataframe for each of the above parameters.

Usage

```
airnow_createDataDataframes(parameters = NULL,  
  startdate = strftime(lubridate::now(), "%Y%m%d00", tz = "UTC"),  
  hours = 24)
```

Arguments

parameters	Vector of names of desired pollutants or NULL for all pollutants.
startdate	Desired start date (integer or character representing YYYYMMDD[HH]).
hours	Desired number of hours of data to assemble.

Value

List of dataframes where each dataframe contains all data for a unique parameter (e.g: "PM2.5", "NOX").

Note

As of 2016-12-27, it appears that hourly data are available only for 2016 and not for earlier years.

See Also

airnow_downloadParseData
airnow_qualityControl

Examples

```
## Not run:
airnowList <- airnow_createDataDataframes("PM2.5", 2019062500)

## End(Not run)
```

```
airnow_createMetaDataframes
```

Create dataframes of AirNow site location metadata

Description

The `airnow_createMetaDataframes()` function uses the `airnow_downloadSites()` function to download site metadata from AirNow and restructures that data into a format that is compatible with the PWFSLSmoke package `ws_monitor` data model.

The meta dataframe in the `ws_monitor` data model has metadata associated with monitoring site locations for a specific parameter and must contain at least the following columns:

- monitorID – per deployment unique ID
- longitude – decimal degrees E
- latitude – decimal degrees N
- elevation – height above sea level in meters
- timezone – olson timezone
- countryCode – ISO 3166-1 alpha-2
- stateCode – ISO 3166-2 alpha-2

The meta dataframe will have rownames matching `monitorID`.

This function takes a dataframe obtained from AirNowTech's `monitoring_site_locations.dat` file, splits it up into separate dataframes, one for each parameter, and performs the following cleanup:

- convert incorrect values to NA e.g. `longitude=0` & `latitude=0`
- add timezone information

Parameters included in AirNow data include at least the following list:

1. BARPR
2. BC
3. CO
4. NO
5. NO2
6. NO2Y
7. NO2X
8. NOX
9. NOOY
10. OC
11. OZONE
12. PM10
13. PM2.5
14. PRECIP
15. RHUM
16. SO2
17. SRAD
18. TEMP
19. UV-AETH
20. WD
21. WS

Setting `parameters=NULL` will generate a separate dataframe for each of the above parameters.

Usage

```
airnow_createMetaDataframes(parameters = NULL,
                             pwfslDataIngestSource = "AIRNOW", addGoogleMeta = TRUE)
```

Arguments

`parameters` vector of names of desired pollutants or NULL for all pollutants

`pwfslDataIngestSource`
 identifier for the source of monitoring data, e.g. 'AIRNOW'

`addGoogleMeta`
 logical specifying wheter to use Google elevation and reverse geocoding services

Value

List of 'meta' dataframes with site metadata for unique parameters (e.g: "PM2.5", "NOX").

See Also

airnow_downloadSites

Examples

```
## Not run:  
metaList <- airnow_createMetaDataframes(parameters = "PM2.5")  
  
## End(Not run)
```

airnow_createMonitorObjects

Obtain AirNow data and create ws_monitor objects

Description

This function uses the `airnow_downloadParseData` function to download monthly dataframes of AirNow data and restructures that data into a format that is compatible with the `PWFSLSmoke` package `ws_monitor` data model.

AirNow data parameters include at least the following list:

1. BARPR
2. BC
3. CO
4. NO
5. NO2
6. NO2Y
7. NO2X
8. NOX
9. NOOY
10. OC
11. OZONE
12. PM10
13. PM2.5
14. PRECIP
15. RHUM
16. SO2
17. SRAD

- 18. TEMP
- 19. UV-AETH
- 20. WD
- 21. WS

Setting `parameters=NULL` will generate a separate `ws_monitor` object for each of the above parameters.

Usage

```
airnow_createMonitorObjects(parameters = NULL,
  startdate = strftime(lubridate::now(), "%Y%m%d", tz = "UTC"),
  hours = 24, zeroMinimum = TRUE, addGoogleMeta = TRUE)
```

Arguments

<code>parameters</code>	vector of names of desired pollutants or NULL for all pollutants
<code>startdate</code>	desired start date (integer or character representing YYYYMMDD[HH])
<code>hours</code>	desired number of hours of data to assemble
<code>zeroMinimum</code>	logical specifying whether to convert negative values to zero
<code>addGoogleMeta</code>	logical specifying wheter to use Google elevation and reverse geocoding services

Value

List where each element contains a `ws_monitor` object for a unique parameter (e.g: "PM2.5", "NOX").

Note

As of 2017-12-17, it appears that hourly data are available only for 2016 and not for earlier years.

See Also

`airnow_createDataDataframes`
`airnow_createMetaDataframes`

Examples

```
## Not run:
monList <- airnow_createMonitorObjects(c("PM2.5"), 20190625)
pm25 <- monList$PM2.5
o3 <- monList$o3

## End(Not run)
```

```
airnow_downloadHourlyData
```

Download hourly data from AirNow

Description

The <https://airnowtech.org> site provides both air pollution monitoring data as well as monitoring site location metadata. This function retrieves a single, hourly data file and returns it as a dataframe.

Usage

```
airnow_downloadHourlyData(datestamp = strftime(lubridate::now(),
  "%Y%m%d00", tz = "UTC"),
  baseUrl = "https://files.airnowtech.org/airnow")
```

Arguments

<code>datestamp</code>	Integer or character representing YYYYMMDDHH.
<code>baseUrl</code>	Base URL for archived hourly data.

Value

Dataframe of AirNow hourly data.

Note

As of 2016-12-27, it appears that hourly data are available only for 2016 and not for earlier years.

Data from locations whose timezones have a fractional offset from UTC are removed as the PWF-SLSmoke data model only supports data reported on hour boundaries. As of 2019-06-26, this only applies to US Department of State monitors in Myanmar, Sri Lanka, India and Nepal.

See Also

```
airnow_createDataDataframes
airnow_downloadParseData
```

Examples

```
## Not run:
df <- airnow_downloadHourlyData(2018070112)

## End(Not run)
```

`airnow_downloadParseData`*Download and aggregate multiple hourly data files from AirNow*

Description

This function makes repeated calls to `airnow_downloadHourlyData` to obtain data from AirNow. All data obtained are then combined into a single tibble and returned.

Parameters included in AirNow data include at least the following list:

1. BARPR
2. BC
3. CO
4. NO
5. NO2
6. NO2Y
7. NO2X
8. NOX
9. NOOY
10. OC
11. OZONE
12. PM10
13. PM2.5
14. PRECIP
15. RHUM
16. SO2
17. SRAD
18. TEMP
19. UV-AETH
20. WD
21. WS

Passing a vector of one or more of the above names as the `parameters` argument will cause the resulting tibble to be filtered to contain only records for those parameters.

Usage

```
airnow_downloadParseData(parameters = NULL,  
  startdate = strftime(lubridate::now(), "%Y%m%d00", tz = "UTC"),  
  hours = 24)
```

Arguments

parameters	vector of names of desired pollutants or NULL for all pollutants
startdate	desired start date (integer or character representing YYYYMMDD[HH])
hours	desired number of hours of data to assemble

Value

Tibble of aggregated AirNow data.

Note

As of 2016-12-27, it appears that hourly data are available only for 2016 and not for earlier years.

See Also

airnow_createDataDataframes
airnow_downloadHourlyData

Examples

```
## Not run:
tbl <- airnow_downloadParseData("PM2.5", 2016070112, hours = 24)

## End(Not run)
```

airnow_downloadSites

Download AirNow Site Location Metadata

Description

The <https://airnowtech.org> site provides both air pollution monitoring data as well as monitoring site location metadata. This function retrieves the most recent version of the site location metadata file and returns it as a dataframe.

A description of the data format is publicly available at the Monitoring Site Fact Sheet¹.

Usage

```
airnow_downloadSites(baseUrl = "https://files.airnowtech.org/airnow/today/",
file = "monitoring_site_locations.dat")
```

Arguments

baseUrl	location of the AirNow monitoring site locations file
file	name of the AirNow monitoring site locations file

¹<https://www.airnowapi.org/docs/MonitoringSiteFactSheet.pdf>

Value

Tibble of site location metadata.

Note

As of December, 2016, the `monitoring_site_locations.dat` file has an encoding of "CP437" (aka "Non-ISO extended-ASCII" or "IBMPC 437") and will be converted to "UTF-8" so that French and Spanish language place names are properly encoded in the returned dataframe.

See Also

`airnow_createMetaDataframes`

Examples

```
## Not run:  
sites <- airnow_downloadSites()  
  
## End(Not run)
```

airnow_load

Load Processed AirNow Monitoring Data

Description

Please use `airnow_loadAnnual` instead of this function. It will soon be deprecated.

Usage

```
airnow_load(year = 2017, month = NULL, parameter = "PM2.5",  
            baseUrl = "https://haze.airfire.org/monitoring/AirNow/RData/")
```

Arguments

<code>year</code>	desired year (integer or character representing YYYY)
<code>month</code>	desired month (integer or character representing MM)
<code>parameter</code>	parameter of interest
<code>baseUrl</code>	base URL for AirNow meta and data files

Value

A `ws_monitor` object with AirNow data.

airnow_loadAnnual *Load annual AirNow monitoring data*

Description

Loads pre-generated .RData files containing annual AirNow data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The annual files loaded by this function are updated on the 15th of each month and cover the period from the beginning of the year to the end of the last month.

For data during the last 45 days, use `airnow_loadDaily()`.

For the most recent data, use `airnow_loadLatest()`.

AirNow parameters include the following:

1. PM2.5

Available AirNow RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/AirNow/RData>²

Usage

```
airnow_loadAnnual(year = NULL, parameter = "PM2.5",  
  baseUrl = "https://haze.airfire.org/monitoring", dataDir = NULL)
```

Arguments

<code>year</code>	Desired year (integer or character representing YYYY).
<code>parameter</code>	Parameter of interest.
<code>baseUrl</code>	Base URL for 'annual' AirNow data files.
<code>dataDir</code>	Local directory containing 'annual' data files.

Value

A `ws_monitor` object with AirNow data.

See Also

`airnow_loadDaily`
`airnow_loadLatest`

²<https://haze.airfire.org/monitoring/AirNow/RData>

Examples

```
## Not run:
airnow_loadAnnual(2017) %>%
  monitor_subset(stateCodes='MT', tlim=c(20170701,20170930)) %>%
  monitor_dailyStatistic() %>%
  monitor_timeseriesPlot(style = 'gnats', ylim=c(0,300), xpd=NA)
  addAQIStackedBar()
  addAQILines()
  title("Montana 2017 -- AirNow Daily Average PM2.5")

## End(Not run)
```

airnow_loadDaily *Load recent AirNow monitoring data*

Description

Loads pre-generated .RData files containing recent AirNow data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The daily files loaded by this function are updated once a day, shortly after midnight and contain data for the previous 45 days.

For the most recent data, use `airnow_loadLatest()`.

For data extended more than 45 days into the past, use `airnow_loadAnnual()`.

AirNow parameters include the following:

1. PM2.5

Available AirNow RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/AirNow/RData/latest>³

Usage

```
airnow_loadDaily(parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL)
```

Arguments

<code>parameter</code>	Parameter of interest.
<code>baseUrl</code>	Base URL for 'daily' AirNow data files.
<code>dataDir</code>	Local directory containing 'daily' data files.

Value

A `ws_monitor` object with AirNow data.

³<https://haze.airfire.org/monitoring/AirNow/RData/latest>

See Also

```
airnow_loadAnnual
airnow_loadLatest
```

Examples

```
## Not run:
airnow_loadDaily() %>%
  monitor_subset(stateCodes=CONUS) %>%
  monitor_map()

## End(Not run)
```

```
airnow_loadLatest Load most recent AirNow monitoring data
```

Description

Loads pre-generated .RData files containing the most recent AirNow data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The files loaded by this function are updated multiple times an hour and contain data for the previous 10 days.

For daily updates covering the most recent 45 days, use `airnow_loadDaily()`.

For data extended more than 45 days into the past, use `airnow_loadAnnual()`.

AirNow parameters include the following:

1. PM2.5

Avaiable RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/AirNow/RData/latest>⁴

Usage

```
airnow_loadLatest(parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL)
```

Arguments

<code>parameter</code>	Parameter of interest.
<code>baseUrl</code>	Base URL for 'daily' AirNow data files.
<code>dataDir</code>	Local directory containing 'daily' data files.

⁴<https://haze.airfire.org/monitoring/AirNow/RData/latest>

Value

A *ws_monitor* object with AirNow data.

See Also

```
airnow_loadAnnual  
airnow_loadDaily
```

Examples

```
## Not run:  
airnow_loadLatest() %>%  
  monitor_subset(stateCodes=CONUS) %>%  
  monitor_map()  
  
## End(Not run)
```

```
airnow_qualityControl  
  Apply Quality Control to AirNow dataframe
```

Description

Perform range validation on AirNow data. This function also replaces values of -999 with NA.

Usage

```
airnow_qualityControl(df, limits = c(-Inf, Inf))
```

Arguments

<code>df</code>	multi-site restructured dataframe created within <code>airnow_createDataDataframe()</code>
<code>limits</code>	lo and hi range of valid values

Value

Cleaned up dataframe of AIRSIS monitor data.

See Also

```
airnow_createDataDataframes
```

AIRSIS

AIRSIS Unit Types

Description

AIRSIS provides access to data by unit type at URLs like: <http://usfs.airsis.com/vision/common/CSVExport.aspx?utid=38&S11-06&EndDate=2017-11-07>

The AIRSIS object is a list of lists. The element named `unitTypes` is itself a list of named unit types:

Unit types include:

- DATARAM 21 = Dataram
- BAM1020 24 = Bam 1020
- EBAM_NEW 30 = eBam-New
- EBAM 38 = Iridium - Ebam
- ESAM 39 = Iridium - Esam
- AUTOMET 43 = Automet

Usage

AIRSIS

Format

A list of lists

Details

AIRSIS monitor types and codes

Note

This list of monitor types was created on Feb 09, 2017.

```
airsis_availableUnits
```

Get AIRSIS available unit identifiers

Description

Returns a list of unitIDs with data during a particular time period.

Usage

```
airsis_availableUnits(startdate = strptime(lubridate::now("UTC"),
"%Y010100", tz = "UTC"), enddate = strptime(lubridate::now("UTC"),
"%Y%m%d23", tz = "UTC"), provider = "USFS",
unitTypes = c("BAM1020", "EBAM", "ESAM"),
baseUrl = "http://xxxx.airsis.com/vision/common/CSVExport.aspx?")
```

Arguments

startdate	desired start date (integer or character representing YYYYMMDD[HH])
enddate	desired end date (integer or character representing YYYYMMDD[HH])
provider	identifier used to modify baseURL ['APCD' 'USFS']
unitTypes	vector of unit types
baseUrl	base URL for data queries

Value

Vector of AIRSIS unitIDs.

References

Interagency Real Time Smoke Monitoring⁵

Examples

```
## Not run:
unitIDs <- aairsis_availableUnits(20150701, 20151231,
                                provider = 'USFS',
                                unitTypes = c('EBAM', 'ESAM'))

## End(Not run)
```

⁵<http://usfs.airsis.com>

```
airsis_BAM1020QualityControl
```

Apply Quality Control to raw AIRSIS BAM1020 dataframe

Description

Perform various QC measures on AIRSIS BAM1020 data.

A POSIXct datetime column (UTC) is also added based on DateTime.

Usage

```
airsis_BAM1020QualityControl(tbl, valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90), remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE, valid_Flow = c(0.834 * 0.95, 0.834 * 1.05),
  valid_AT = c(-Inf, 45), valid_RHi = c(-Inf, 45),
  valid_Conc = c(-Inf, 5000), flagAndKeep = FALSE)
```

Arguments

<code>tbl</code>	single site tibble created by <code>airsis_parseData()</code>
<code>valid_Longitude</code>	range of valid Longitude values
<code>valid_Latitude</code>	range of valid Latitude values
<code>remove_Lon_zero</code>	flag to remove rows where Longitude == 0
<code>remove_Lat_zero</code>	flag to remove rows where Latitude == 0
<code>valid_Flow</code>	range of valid Flow values
<code>valid_AT</code>	range of valid AT values
<code>valid_RHi</code>	range of valid RHi values
<code>valid_Conc</code>	range of valid ConcHr values
<code>flagAndKeep</code>	flag, rather than remove, bad data during the QC process

Value

Cleaned up tibble of AIRSIS monitor data.

See Also

`airsis_qualityControl`

```
airsis_createDataDataframe
      Create AIRSIS data dataframe
```

Description

After quality control has been applied to an AIRSIS tibble, we can extract the PM2.5 values and store them in a `data` dataframe organized as time-by-deployment (aka time-by-site).

The first column of the returned dataframe is named `'datetime'` and contains a `POSIXct` time in UTC. Additional columns contain data for each separate deployment of a monitor.

Usage

```
airsis_createDataDataframe(tbl, meta)
```

Arguments

<code>tbl</code>	single site AIRSIS tibble created by <code>airsis_clustering()</code>
<code>meta</code>	AIRSIS meta dataframe created by <code>airsis_createMetaDataframe()</code>

Value

A `data` dataframe for use in a `ws_monitor` object.

```
airsis_createMetaDataframe
      Create AIRSIS site location metadata dataframe
```

Description

After an AIRSIS tibble has been enhanced with additional columns generated by `addClustering` we are ready to pull out site information associated with unique deployments.

These will be rearranged into a dataframe organized as deployment-by-property with one row for each monitor deployment.

This site information found in `tbl` is augmented so that we end up with a uniform set of properties associated with each monitor deployment. The list of columns in the returned `meta` dataframe is:

```
> names(p$meta)
 [1] "monitorID"           "longitude"           "latitude"
 [4] "elevation"           "timezone"             "countryCode"
 [7] "stateCode"           "siteName"             "agencyName"
[10] "countyName"          "msaName"              "monitorType"
[13] "monitorInstrument"   "aqslID"               "pwfslID"
[16] "pwfslDataIngestSource" "telemetryAggregator" "telemetryUnitID"
```

Usage

```
airsis_createMetaDataframe(tbl, provider = as.character(NA),
  unitID = as.character(NA), pwfslDataIngestSource = "AIRSIS",
  existingMeta = NULL, addGoogleMeta = FALSE, addEsriMeta = FALSE)
```

Arguments

tbl	single site AIRSIS tibble after metadata enhancement
provider	identifier used to modify baseURL ['APCD' 'USFS']
unitID	character or numeric AIRSIS unit identifier
pwfslDataIngestSource	identifier for the source of monitoring data, e.g. 'AIRSIS'
existingMeta	existing 'meta' dataframe from which to obtain metadata for known monitor deployments
addGoogleMeta	logical specifying wheter to use Google elevation and reverse geocoding services
addEsriMeta	logical specifying wheter to use ESRI elevation and reverse geocoding services

Value

A meta dataframe for use in a *ws_monitor* object.

See Also

addMazamaMetadata

airsis_createMonitorObject

Obtain AIRSIS data and create ws_monitor object

Description

Obtains monitor data from an AIRSIS webservice and converts it into a quality controlled, metadata enhanced *ws_monitor* object ready for use with all *monitor_~* functions.

Steps involved include:

1. download CSV text
2. parse CSV text
3. apply quality control
4. apply clustering to determine unique deployments
5. enhance metadata to include: elevation, timezone, state, country, site name
6. reshape AIRSIS data into deployment-by-property *meta* and and time-by-deployment *data* dataframes

QC parameters that can be passed in the ... include the following valid data ranges as taken from `airsis_EBAMQualityControl()`:

- `valid_Longitude=c(-180,180)`
- `valid_Latitude=c(-90,90)`
- `remove_Lon_zero = TRUE`
- `remove_Lat_zero = TRUE`
- `valid_Flow = c(16.7*0.95,16.7*1.05)`
- `valid_AT = c(-Inf,45)`
- `valid_RHi = c(-Inf,45)`
- `valid_Conc = c(-Inf,5.000)`

Note that appropriate values for QC thresholds will depend on the type of monitor.

Usage

```
airsis_createMonitorObject(startdate = strftime(lubridate::now(),
"%Y010100", tz = "UTC"), enddate = strftime(lubridate::now(),
"%Y%m%d23", tz = "UTC"), provider = NULL, unitID = NULL,
clusterDiameter = 1000, zeroMinimum = TRUE,
baseUrl = "http://xxxx.airsis.com/vision/common/CSVExport.aspx?",
saveFile = NULL, existingMeta = NULL, addGoogleMeta = FALSE,
addEsriMeta = FALSE, ...)
```

Arguments

<code>startdate</code>	desired start date (integer or character representing YYYYMMDD[HH])
<code>enddate</code>	desired end date (integer or character representing YYYYMMDD[HH])
<code>provider</code>	identifier used to modify baseURL ['APCD' 'USFS']
<code>unitID</code>	character or numeric AIRSIS unit identifier
<code>clusterDiameter</code>	diameter in meters used to determine the number of clusters (see <code>addClustering()</code>)
<code>zeroMinimum</code>	logical specifying whether to convert negative values to zero
<code>baseUrl</code>	base URL for data queries
<code>saveFile</code>	optional filename where raw CSV will be written
<code>existingMeta</code>	existing 'meta' dataframe from which to obtain metadata for known monitor deployments
<code>addGoogleMeta</code>	logical specifying wheter to use Google elevation and reverse geocoding services
<code>addEsriMeta</code>	logical specifying wheter to use ESRI elevation and reverse geocoding services
...	additional parameters are passed to type-specific QC functions

Value

A *ws_monitor* object with AIRSIS data.

Note

The downloaded CSV may be saved to a local file by providing an argument to the `saveFile` parameter.

See Also

```
airsis_downloadData
airsis_parseData
airsis_qualityControl
addClustering
airsis_createMetaDataframe
airsis_createDataDataframe
```

Examples

```
## Not run:
initializeMazamaSpatialUtils()
usfs_1013 <- aairsis_createMonitorObject(20150301, 20150831, 'USFS', unitID='1013')
monitor_leaflet(usfs_1013)

## End(Not run)
```

```
airsis_createRawDataframe
```

Obtain AIRSIS data and parse into a raw tibble

Description

Obtains monitor data from an AIRSIS webservice and converts it into a quality controlled, metadata enhanced "raw" tibble ready for use with all `raw_~` functions.

Steps involved include:

1. download CSV text
2. parse CSV text
3. apply quality control
4. apply clustering to determine unique deployments
5. enhance metadata to include: elevation, timezone, state, country, site name

Usage

```
airsis_createRawDataframe(startdate = strptime(lubridate::now(),
"%Y010100", tz = "UTC"), enddate = strptime(lubridate::now(),
"%Y%m%d23", tz = "UTC"), provider = NULL, unitID = NULL,
clusterDiameter = 1000,
baseUrl = "http://xxxx.airsis.com/vision/common/CSVExport.aspx?",
saveFile = NULL, flagAndKeep = FALSE)
```

Arguments

startdate	Desired start date (integer or character representing YYYYMMDD[HH]).
enddate	Desired end date (integer or character representing YYYYMMDD[HH]).
provider	Identifier used to modify baseURL ['APCD' 'USFS'].
unitID	Character or numeric AIRSIS unit identifier.
clusterDiameter	Diameter in meters used to determine the number of clusters (see addClustering).
baseUrl	Base URL for data queries.
saveFile	Optional filename where raw CSV will be written.
flagAndKeep	Flag, rather than remove, bad data during the QC process.

Value

Raw tibble of AIRSIS data.

Note

The downloaded CSV may be saved to a local file by providing an argument to the `saveFile` parameter.

See Also

```
airsis_downloadData
airsis_parseData
airsis_qualityControl
addClustering
```

Examples

```
## Not run:
raw <- aairsis_createRawDataframe(startdate = 20160901,
                                provider = 'USFS',
                                unitID = '1033')

raw <- raw_enhance(raw)
rawPlot_timeseries(raw, tlim = c(20160908,20160917))

## End(Not run)
```

`airsis_downloadData`*Download AIRSIS data*

Description

Request data from a particular station for the desired time period. Data are returned as a single character string containing the AIRIS output.

Usage

```
airsis_downloadData(startdate = strftime(lubridate::now(), "%Y0101", tz =
  "UTC"), enddate = strftime(lubridate::now(), "%Y%m%d", tz =
  "UTC"), provider = "USFS", unitID = NULL,
  baseUrl = "http://xxxx.airsis.com/vision/common/CSVExport.aspx?")
```

Arguments

<code>startdate</code>	desired start date (integer or character representing YYYYMMDD[HH])
<code>enddate</code>	desired end date (integer or character representing YYYYMMDD[HH])
<code>provider</code>	identifier used to modify baseURL ['APCD' 'USFS']
<code>unitID</code>	unit identifier
<code>baseUrl</code>	base URL for data queries

Value

String containing AIRSIS output.

References

Interagency Real Time Smoke Monitoring⁶

Examples

```
## Not run:
fileString <- aairsis_downloadData( 20150701, 20151231, provider='USFS', unitID='1026')
df <- aairsis_parseData(fileString)

## End(Not run)
```

⁶<http://usfs.airsis.com>

```
airsis_EBAMQualityControl
```

Apply Quality Control to raw AIRSIS EBAM tibble

Description

Perform various QC measures on AIRSIS EBAM data.

The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on Date.Time.GMT.

Usage

```
airsis_EBAMQualityControl(tbl, valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90), remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE, valid_Flow = c(16.7 * 0.95, 16.7 * 1.05),
  valid_AT = c(-Inf, 45), valid_RHi = c(-Inf, 45),
  valid_Conc = c(-Inf, 5), flagAndKeep = FALSE)
```

Arguments

<code>tbl</code>	single site tibble created by <code>airsis_parseData()</code>
<code>valid_Longitude</code>	range of valid Longitude values
<code>valid_Latitude</code>	range of valid Latitude values
<code>remove_Lon_zero</code>	flag to remove rows where Longitude == 0
<code>remove_Lat_zero</code>	flag to remove rows where Latitude == 0
<code>valid_Flow</code>	range of valid Flow values
<code>valid_AT</code>	range of valid AT values
<code>valid_RHi</code>	range of valid RHi values
<code>valid_Conc</code>	range of valid ConcHr values
<code>flagAndKeep</code>	flag, rather than remove, bad data during the QC process

Value

Cleaned up tibble of AIRSIS monitor data.

See Also

airsis_qualityControl

airsis_ESAMQualityControl

Apply Quality Control to raw AIRSIS E-Sampler dataframe

Description

Perform various QC measures on AIRSIS E-Sampler data.

The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on TimeStamp.

Usage

```
airsis_ESAMQualityControl(tbl, valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90), remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE, valid_Flow = c(1.999, 2.001),
  valid_AT = c(-Inf, 150), valid_RHi = c(-Inf, 55),
  valid_Conc = c(-Inf, 5000), flagAndKeep = FALSE)
```

Arguments

tbl	single site tibble created by <code>airsis_downloadData()</code>
valid_Longitude	range of valid Longitude values
valid_Latitude	range of valid Latitude values
remove_Lon_zero	flag to remove rows where Longitude == 0
remove_Lat_zero	flag to remove rows where Latitude == 0
valid_Flow	range of valid Flow.l.m values
valid_AT	range of valid AT.C. values
valid_RHi	range of valid RHi... values
valid_Conc	range of valid Conc.mg.m3. values
flagAndKeep	flag, rather than remove, bad data during the QC process

Value

Cleaned up tibble of AIRSIS monitor data.

See Also

`airsis_qualityControl`

`airsis_identifyMonitorType`

Identify AIRSIS monitor type

Description

Examine the column names of the incoming dataframe (or first line of raw text) to identify different types of monitor data provided by AIRSIS.

The return is a list includes everything needed to identify and parse the raw data using `readr::read_csv()`:

- `monitorType` – identification string
- `rawNames` – column names from the data (including special characters)
- `columnNames` – assigned column names (special characters repaced with '.')
- `columnTypes` – column type string for use with `readr::read_csv()`

The `monitorType` will be one of:

- "BAM1020" – BAM1020 (e.g. USFS #49 in 2010)
- "EBAM" – EBAM (e.g. USFS #1026 in 2010)
- "ESAM" – E-Sampler (e.g. USFS #1002 in 2010)
- "UNKOWN" – ???

Usage

```
airsis_identifyMonitorType(df)
```

Arguments

`df` dataframe or raw character string containing AIRSIS data

Value

List including `monitorType`, `rawNames`, `columnNames` and `columnTypes`.

References

Interagency Real Time Smoke Monitoring⁷

⁷<http://usfs.airsis.com>

Examples

```
## Not run:
fileString <- aairsis_downloadData( 20150701, 20151231, provider='USFS', unitID='1026')
monitorTypeList <- aairsis_identifyMonitorType(fileString)

## End(Not run)
```

airsis_load *Load Processed AIRSIS Monitoring Data*

Description

Please use `airsis_loadAnnual` instead of this function. It will soon be deprecated.

Usage

```
airsis_load(year = 2017,
            baseUrl = "https://haze.airfire.org/monitoring/AIRSIS/RData/")
```

Arguments

<code>year</code>	desired year (integer or character representing YYYY)
<code>baseUrl</code>	base URL for AIRSIS meta and data files

Value

A `ws_monitor` object with AIRSIS data.

`airsis_loadAnnual` *Load annual AIRSIS monitoring data*

Description

Loads pre-generated .RData files containing annual AIRSIS data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The annual files loaded by this function are updated on the 15th of each month and cover the period from the beginning of the year to the end of the last month.

For data during the last 45 days, use `airsis_loadDaily()`.

For the most recent data, use `airsis_loadLatest()`.

AIRSIS parameters include the following:

1. PM2.5

Available AIRSIS RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/AIRSIS/RData>⁸

⁸<https://haze.airfire.org/monitoring/AIRSIS/RData>

Usage

```
airsis_loadAnnual(year = NULL, parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring", dataDir = NULL)
```

Arguments

year	Desired year (integer or character representing YYYY).
parameter	Parameter of interest.
baseUrl	Base URL for 'annual' AIRSIS data files.
dataDir	Local directory containing 'annual' data files.

Value

A *ws_monitor* object with AIRSIS data.

See Also

```
airsis_loadDaily
airsis_loadLatest
```

Examples

```
## Not run:
airsis_loadAnnual(2017) %>%
  monitor_subset(stateCodes='MT', tlim=c(20170701,20170930)) %>%
  monitor_dailyStatistic() %>%
  monitor_timeseriesPlot(style = 'gnats', ylim=c(0,300), xpd=NA)
  addAQIStackedBar()
  addAQIILines()
  title("Montana 2017 -- AIRSIS Daily Average PM2.5")

## End(Not run)
```

airsis_loadDaily *Load recent AIRSIS monitoring data*

Description

Loads pre-generated .RData files containing recent AIRSIS data.

If *dataDir* is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The daily files loaded by this function are updated once a day, shortly after midnight and contain data for the previous 45 days.

For the most recent data, use `airsis_loadLatest()`.

For data extended more than 45 days into the past, use `airsis_loadAnnual()`.

AIRSIS parameters include the following:

1. PM2.5

Avaiable AIRSIS RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/AIRSIS/RData/latest>⁹

Usage

```
airsis_loadDaily(parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL)
```

Arguments

parameter	Parameter of interest.
baseUrl	Base URL for 'daily' AirNow data files.
dataDir	Local directory containing 'daily' data files.

Value

A *ws_monitor* object with AIRSIS data.

See Also

```
airsis_loadAnnual
airsis_loadLatest
```

Examples

```
## Not run:
airsis_loadDaily() %>%
  monitor_subset(stateCodes=CONUS) %>%
  monitor_map()

## End(Not run)
```

`airsis_loadLatest` *Load most recent AIRSIS monitoring data*

Description

Loads pre-generated .RData files containing the most recent AIRSIS data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The files loaded by this function are updated multiple times an hour and contain data for the previous 10 days.

For daily updates covering the most recent 45 days, use `airsis_loadDaily()`.

For data extended more than 45 days into the past, use `airsis_loadAnnual()`.

AIRSIS parameters include the following:

⁹<https://haze.airfire.org/monitoring/AIRSIS/RData/latest>

1. PM2.5

Avaiable RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/AIRSIS/RData/latest>¹⁰

Usage

```
airsis_loadLatest(parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL)
```

Arguments

parameter	Parameter of interest.
baseUrl	Base URL for 'daily' AirNow data files.
dataDir	Local directory containing 'daily' data files.

Value

A *ws_monitor* object with AIRSIS data.

See Also

```
airsis_loadAnnual
airsis_loadDaily
```

Examples

```
## Not run:
airsis_loadLatest() %>%
  monitor_subset(stateCodes=CONUS) %>%
  monitor_map()

## End(Not run)
```

airsis_parseData *Parse AIRSIS data string*

Description

Raw character data from AIRSIS are parsed into a tibble. The incoming `fileString` can be read in directly from AIRSIS using `airsis_downloadData()` or from a local file using `readr::read_file()`.

The type of monitor represented by this `fileString` is inferred from the column names using `airsis_identifyMonitorT` and appropriate column types are assigned. The character data are then read into a tibble and augmented in the following ways:

1. Longitude, Latitude and any System Voltage values, which are only present in GPS timestamp rows, are propagated foward using a last-observation-carry-forward algorithm'

¹⁰<https://haze.airfire.org/monitoring/AIRSIS/RData/latest>

2. Longitude, Latitude and any System Voltage values, which are only present in GPS timestamp rows, are propagated backwards using a first-observation-carry-backward algorithm'
3. GPS timestamp rows are removed'

Usage

```
airsis_parseData(fileString)
```

Arguments

`fileString` character string containing AIRSIS data as a csv

Value

Dataframe of AIRSIS raw monitor data.

References

Interagency Real Time Smoke Monitoring¹¹

Examples

```
## Not run:
library(MazamaWebUtils)
fileString <- aairsis_downloadData(20150701, 20151231, provider='USFS', unitID='1026')
tbl <- aairsis_parseData(fileString)

## End(Not run)
```

```
airsis_qualityControl
```

Apply Quality Control to raw AIRSIS dataframe

Description

Various QC steps are taken to clean up the incoming raw tibble including:

1. Ensure GPS location data are included in each measurement record.
2. Remove GPS location records.
3. Remove measurement records with values outside of valid ranges.

See the individual `airsis_~QualityControl()` functions for details.

QC parameters that can be passed in the `...` include the following valid data ranges as taken from `airsis_EBAMQualityControl()`:

- `valid_Longitude=c(-180,180)`

¹¹<http://usfs.airsis.com>

- `valid_Latitude=c(-90, 90)`
- `remove_Lon_zero = TRUE`
- `remove_Lat_zero = TRUE`
- `valid_Flow = c(16.7*0.95, 16.7*1.05)`
- `valid_AT = c(-Inf, 45)`
- `valid_RHi = c(-Inf, 45)`
- `valid_Conc = c(-Inf, 5.000)`

Note that appropriate values for QC thresholds will depend on the type of monitor.

Usage

```
airsis_qualityControl(tbl, ...)
```

Arguments

<code>tbl</code>	single site tibble created by <code>airsis_downloadData()</code>
<code>...</code>	additional parameters are passed to type-specific QC functions

Value

Cleaned up tibble of AIRSIS monitor data.

See Also

```
airsis_EBAMQualityControl
airsis_ESAMQualityControl
```

AQI

Official Air Quality Index Levels, Names and Colors

Description

Official AQI levels, names and colors are provided in a list for easy coloring and labeling.

Usage

```
AQI
```

Format

A list with named elements

Details

AQI breaks and associated names and colors

The AQI object contains english language text.

AQI breaks and colors are defined at <https://docs.airnowapi.org/aq101>

Note

The low end of each break category is used as the breakpoint.

See Also

AQI_en AQI_es

aqiColors

Generate AQI Colors

Description

This function uses the `leaflet::colorBin()` function to return a vector or matrix of colors derived from PM2.5 values.

Usage

```
aqiColors(x, palette = AQI$colors, domain = c(0, 1e+06),
  bins = AQI$breaks_24, na.color = NA)
```

Arguments

<code>x</code>	vector or matrix of PM2.5 values or a <i>ws_monitor</i> object
<code>palette</code>	color palette (see <code>leaflet::colorBin()</code>)
<code>domain</code>	range of valid data (see <code>leaflet::colorBin()</code>)
<code>bins</code>	color bins (see <code>leaflet::colorBin()</code>)
<code>na.color</code>	missing value color (see <code>leaflet::colorBin()</code>)

Value

A vector or matrix of AQI colors to be used in maps and plots.

Examples

```

wa <- monitor_subset(Northwest_Megafires, stateCodes='WA', tlim=c(20150821,20150828))
colorMatrix <- aqiColors(wa)
time <- wa$data$datetime
pm25 <- wa$data[, -1]
plot(time, pm25[,1], col=colorMatrix[,1],
      ylim=range(pm25, na.rm=TRUE),
      xlab="2015", ylab="PM 2.5 (ug/m3)", main="Washington State Smoke")
for ( i in seq_along(pm25) ) {
  points(time, pm25[,i], col=colorMatrix[,i], pch=16)
}

```

aqiPalette

Color Palettes for Air Quality Monitoring Data

Description

Creates a **leaflet** color palette function that can be used to convert monitoring data into vectors of colors.

Usage

```
aqiPalette(style = "aqi", reverse = FALSE)
```

Arguments

style	Palette style, one of 'aqi'.
reverse	Logical specifying whether the colors (or color function) in palette should be used in reverse order.

Value

A function that takes a single parameter *x*; when called with a vector of numbers, #RRGGBB color strings are returned.

See Also

'leaflet::colorBin()'

Examples

```

pm25 <- PWFSLSmoke::Carmel_Valley$data[,2]
binned_colors <- aqiPalette("aqi")(pm25)
plot(pm25, col=binned_colors, pch=15, main='Binned Colors')

```

`AQI_en`*Official Air Quality Index Levels, Names and Colors*

Description

Official AQI levels, names and colors are provided in a list for easy coloring and labeling.

Usage`AQI_en`**Format**

A list with named elements

Details

AQI breaks and associated names and colors (english language)

The `AQI_es` object contains english language text. It is equivalent to the `AQI` object and provided for consistency with other language versions.

AQI breaks and colors are defined at <https://docs.airnowapi.org/aq101>

Note

The low end of each break category is used as the breakpoint.

See Also`AQI AQI_es`

`AQI_es`*Official Air Quality Index Levels, Names and Colors*

Description

Official AQI levels, names and colors are provided in a list for easy coloring and labeling.

Usage`AQI_es`**Format**

A list with named elements

Details

AQI breaks and associated names and colors (Spanish language)

The `AQI_es` object contains spanish language text.

AQI breaks and colors are defined at <https://docs.airnowapi.org/aq101>

Note

The low end of each break category is used as the breakpoint.

See Also

`AQI_en AQI`

`Carmel_Valley`

Carmel Valley Example Dataset

Description

In August of 2016, the Soberanes fire in California burned along the Big Sur coast. It was at the time the most expensive wildfire in US history. This dataset contains PM2.5 monitoring data for the monitor in Carmel Valley which shows heavy smoke as well as strong diurnal cycles associated with sea breezes. Data are stored as a `ws_monitor` object and are used in some examples in the package documentation.

Format

A list with two elements

Details

Carmel Valley example dataset

`CONUS`

CONUS State Codes

Description

State codes for the 48 contiguous states +DC that make up the CONTinental US

Usage

`CONUS`

Format

A vector with 49 elements

Details

CONUS state codes

distance *Calculate distances between points*

Description

This function uses the Haversine formula for calculating great circle distances between points. This formula is purported to work better than the spherical law of cosines for very short distances.

Usage

```
distance(targetLon, targetLat, longitude, latitude)
```

Arguments

targetLon	longitude (decimal degrees) of the point from which distances are calculated
targetLat	latitude (decimal degrees) of the point from which distances are calculated
longitude	vector of longitudes for which a distance is calculated
latitude	vector of latitudes for which a distance is calculated

Value

Vector of distances in km.

Examples

```
# Seattle to Portland airports
SEA_lon <- -122.3088
SEA_lat <- 47.4502
PDX_lon <- -122.5951
PDX_lat <- 45.5898
distance(SEA_lon, SEA_lat, PDX_lon, PDX_lat)
```

```
epa_createDataDataframe
      Create EPA data dataframe
```

Description

After additional columns(i.e. `datetime`, and `monitorID`) have been applied to an EPA dataframe, we are ready to extract the PM2.5 values and store them in a `data` dataframe organized as time-by-monitor.

The first column of the returned dataframe is named `datetime` and contains a `POSIXct` time in UTC. Additional columns contain data for each separate `monitorID`.

Usage

```
epa_createDataDataframe(tbl)
```

Arguments

`tbl` an EPA raw tibble after metadata enhancement

Value

A `data` dataframe for use in a `ws_monitor` object.

```
epa_createMetaDataframe
      Create dataframe of EPA site location metadata
```

Description

After additional columns(i.e. `datetime`, and `monitorID`) have been applied to an EPA dataframe, we are ready to pull out site information associated with unique `monitorID`.

These will be rearranged into a dataframe organized as deployment-by-property with one row for each `monitorID`.

This site information found in `tbl` is augmented so that we end up with a uniform set of properties associated with each `monitorID`. The list of columns in the returned `meta` dataframe is:

```
> names(p$meta)
 [1] "monitorID"           "longitude"           "latitude"
 [4] "elevation"           "timezone"            "countryCode"
 [7] "stateCode"           "siteName"            "agencyName"
[10] "countyName"          "msaName"             "monitorType"
[13] "monitorInstrument"   "aqslID"              "pwfslID"
[16] "pwfslDataIngestSource" "telemetryAggregator" "telemetryUnitID"
```


Usage

```
epa_createMetaDataframe(tbl, pwfslDataIngestSource = "EPA",
  existingMeta = NULL, addGoogleMeta = TRUE)
```

Arguments

`tbl` an EPA raw tibble after metadata enhancement

`pwfslDataIngestSource` identifier for the source of monitoring data, e.g. 'EPA_hourly_88101_2016.zip'

`existingMeta` existing 'meta' dataframe from which to obtain metadata for known monitor deployments

`addGoogleMeta` logical specifying wheter to use Google elevation and reverse geocoding services

Value

A meta dataframe for use in a `ws_monitor` object.

References

EPA AirData Pre-Generated Data Files¹²
file format description¹³

`epa_createMonitorObject`

Download and convert hourly EPA air quality data

Description

Convert EPA data into a `ws_monitor` object, ready for use with all `monitor_~` functions.

Usage

```
epa_createMonitorObject(zipFile = NULL, zeroMinimum = TRUE,
  addGoogleMeta = TRUE)
```

Arguments

`zipFile` absolute path to monitoring data .zip file

`zeroMinimum` logical specifying whether to convert negative values to zero

`addGoogleMeta` logical specifying wheter to use Google elevation and reverse geocoding services

¹²https://aqs.epa.gov/aqsweb/airdata/download_files.html#Raw

¹³https://aqs.epa.gov/aqsweb/airdata/FileFormats.html#_format_3

Value

A *ws_monitor* object with EPA data.

Note

Before running this function you must first enable spatial data capabilities as in the example.

References

EPA AirData Pre-Generated Data Files¹⁴
file format description¹⁵

Examples

```
## Not run:  
initializeMazamaSpatialUtils()  
zipFile <- epa_downloadData(2016, "88101", downloadDir = '~/Data/EPA')  
mon <- epa_createMonitorObject(zipFile, addGoogleMeta = FALSE)  
  
## End(Not run)
```

epa_downloadData *Download EPA air quality data*

Description

This function downloads air quality data from the EPA and saves it to a directory.

Available parameter codes include:

1. 44201 – Ozone
2. 42401 – SO₂
3. 42101 – CO
4. 42602 – NO₂
5. 88101 – PM_{2.5}
6. 88502 – PM_{2.5}
7. 81102 – PM₁₀
8. SPEC – PM_{2.5}
9. WIND – Wind
10. TEMP – Temperature
11. PRESS – Barometric Pressure
12. RH_DP – RH and dewpoint
13. HAPS – HAPs
14. VOCS – VOCs
15. NONOxNOy

¹⁴https://aqs.epa.gov/aqsweb/airdata/download_files.html#Raw

¹⁵https://aqs.epa.gov/aqsweb/airdata/FileFormats.html#_format_3

Usage

```
epa_downloadData(year = NULL, parameterCode = "88101",
  downloadDir = tempdir(),
  baseUrl = "https://aqs.epa.gov/aqsweb/airdata/")
```

Arguments

```
year          year
parameterCode pollutant code
downloadDir   directoroy where monitoring data .zip file will be saved
baseUrl       base URL for archived daily data
```

Value

Filepath of the downloaded zip file.

Note

Unzipped CSV files are almost 100X larger than the compressed .zip files.

References

EPA AirData Pre-Generated Data Files¹⁶

Examples

```
## Not run:
zipFile <- epa_downloadData(2016, "88101", '~/Data/EPA')
tbl <- epa_parseData(zipFile, "PM2.5")

## End(Not run)
```

epa_load

Load Processed EPA Monitoring Data

Description

Please use `airsis_loadAnnual` instead of this function. It will soon be deprecated.

Usage

```
epa_load(year = strftime(lubridate::now(), "%Y", tz = "UTC"),
  parameterCode = "88101",
  baseUrl = "https://haze.airfire.org/monitoring/EPA/RData/")
```

¹⁶https://aqs.epa.gov/aqsweb/airdata/download_files.html#Raw

Arguments

year desired year (integer or character representing YYYY)
parameterCode pollutant code
baseUrl base URL for EPA .RData files

Value

A *ws_monitor* object with EPA data for an entire year.

epa_loadAnnual *Load annual EPA monitoring data*

Description

Loads pre-generated .RData files containing annual EPA data.

EPA parameter codes include:

1. 88101 – PM2.5 FRM/FEM Mass (begins in 2008)
2. 88502 – PM2.5 non FRM/FEM Mass (begins in 1998)

Available RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/EPA/RData/>¹⁷

Usage

```
epa_loadAnnual(year = NULL, parameterCode = NULL,
               baseUrl = "https://haze.airfire.org/monitoring", dataDir = NULL)
```

Arguments

year Desired year (integer or character representing YYYY).
parameterCode Pollutant code.
baseUrl Base URL for 'annual' EPA data files.
dataDir Local directory containing 'annual' data files.

Value

A *ws_monitor* object with EPA data.

References

EPA AirData Pre-Generated Data Files¹⁸

¹⁷<https://haze.airfire.org/monitoring/EPA/RData/>

¹⁸https://aqs.epa.gov/aqsweb/airdata/download_files.html#Raw

Examples

```
## Not run:
epa_loadAnnual(2000, "88502") %>%
  monitor_subset(stateCodes = 'WA', tlim=c(20000701,20000801)) %>%
  monitor_map()

## End(Not run)
```

epa_parseData *Parse EPA data*

Description

This function uncompress previously downloaded air quality .zip files from the EPA and reads it into a tibble.

Available parameters include:

1. Ozone
2. SO2
3. CO
4. NO2
5. PM2.5
6. PM10
7. Wind
8. Temperature
9. Barometric_Pressure
10. RH_and_Dewpoint
11. HAPs
12. VOCs
13. NONOxNOy

Associated parameter codes include:

1. 44201 – Ozone
2. 42401 – SO2
3. 42101 – CO
4. 42602 – NO2
5. 88101 – PM2.5
6. 88502 – PM2.5
7. 81102 – PM10
8. SPEC – PM2.5

9. WIND – Wind
10. TEMP – Temperature
11. PRESS – Barometric Pressure
12. RH_DP – RH and dewpoint
13. HAPS – HAPs
14. VOCS – VOCs
15. NONOxNOy

Usage

```
epa_parseData(zipFile = NULL)
```

Arguments

`zipFile` absolute path to monitoring data .zip file

Value

Tibble of EPA data.

Note

Unzipped CSV files are almost 100X larger than the compressed .zip files. CSV files are removed after data are read into a dataframe.

References

EPA AirData Pre-Generated Data Files¹⁹

file format description²⁰

Examples

```
## Not run:  
zipFile <- epa_downloadData(2016, "88101", '~/Data/EPA')  
tbl <- epa_parseData(zipFile, "PM2.5")  
  
## End(Not run)
```

¹⁹https://aqs.epa.gov/aqsweb/airdata/download_files.html#Raw

²⁰https://aqs.epa.gov/aqsweb/airdata/FileFormats.html#_format_3

`esriToken`*Token used for ESRI Geocoding Requests*

Description

All package functions that interact with ESRI location services will use the token whenever a request is made.

Format

Character string.

See Also

`addEsriAddress`

`generic_downloadData`*Download generic data*

Description

This function takes a location to a delimited file, gets the file, and returns a string containing the file data.

Usage

```
generic_downloadData(filePath)
```

Arguments

`filePath` Either a path to a file, or a connection (`http(s)://`, `ftp(s)://`).

Details

This function is essentially a wrapper for `read_file`.

Value

A character vector of length 1, containing data from the file located at `filePath`.

Examples

```
## Not run:
# make current directory PWFSLSmoke package directory
filePath <- "../localData/airsis_ebam_example-clean.csv"

fileString <- generic_downloadData(filePath)

## End(Not run)
```

generic_parseData *Parse generic air quality files*

Description

Given a string of delimited file data, this function will parse the file as a table of data and apply some transformations and augmentations as specified by a given configuration list.

Usage

```
generic_parseData(fileString = NULL, configList = NULL)
```

Arguments

fileString Character string of delimited data to parse.

configList A R list or JSON file containing key-value pairs which affect how the parsing of **fileString** is handled. If **configList** is in JSON format, it can be passed in as a file, string, or URL.

Value

A tibble of the data contained in **fileString** parsed according to parameters in **configList**. The data is coerced into a format that is more easily convertible into a **ws_monitor** object at a later point.

Parsing data

Internally, this function uses **read_delim** to convert **fileString** into a tibble. If any lines of data cannot be properly parsed, an error will be thrown and the problem lines will be printed.

Creating a configList

For more information on how to build a **configList**, see the Rmarkdown document "Working with Generic Data" in the **localNotebooks** directory.

Examples

```
filePath <- system.file(
  "extdata", "generic_data_example.csv",
  package = "PWFSLSmoke",
  mustWork = TRUE
)

configPath <- system.file(
  "extdata", "generic_configList_example.json",
  package = "PWFSLSmoke",
  mustWork = TRUE
)

configList <- jsonlite::fromJSON(configPath)
fileString <- generic_downloadData(filePath)
parsedData <- generic_parseData(fileString, configList)
```

`getEsriToken`*Get ESRI Token*

Description

Returns the current esriToken.

Usage

```
getEsriToken()
```

Value

String.

See Also

`addEsriAddress`

`esriToken`

`setEsriToken`

getGoogleApiKey *Get Google API Key*

Description

Returns the current Google API key.

Usage

```
getGoogleApiKey()
```

Value

String.

See Also

addGoogleAddress
addGoogleElevation
googleApiKey
setGoogleApiKey

googleApiKey *API Key used for Google Geocoding Requests*

Description

All package functions that interact with Google location services will use API key whenever a request is made.

Format

Character string.

See Also

addGoogleAddress
addGoogleElevation

```
initializeMazamaSpatialUtils
```

Initialize Mazama Spatial Utils

Description

Convenience function that wraps:

```
logger.setup()
logger.setLevel(WARN)
setSpatialDataDir('~Data/Spatial')
loadSpatialData('NaturalEarthAdm1')
```

If file logging is desired, these commands should be run individually with output log files specified as arguments to `logger.setup()`.

Usage

```
initializeMazamaSpatialUtils(spatialDataDir = "~/Data/Spatial",
                             stateCodeDataset = "NaturalEarthAdm1", logLevel = WARN)
```

Arguments

<code>spatialDataDir</code>	directory where spatial datasets are created
<code>stateCodeDataset</code>	MazamaSpatialUtils dataset returning ISO 3166-2 alpha-2 stateCodes
<code>logLevel</code>	directory where spatial datasets are created

See Also

{[link{logger.setup}](#)}

```
loadDaily
```

Load Recent PM2.5 Monitoring Data

Description

Wrapper function to load and combine recent data from AirNow, AIRSIS and WRCC:

```
airnow <- airnow_loadDaily()
airsis <- aairsis_loadDaily()
wrcc <- wrcc_loadDaily()
ws_monitor <- monitor_combine(list(airnow, aairsis, wrcc))
```

The daily files are generated once a day, shortly after midnight and contain data for the previous 45 days.

For the most recent data, use `loadLatest()`.

Avaiable RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/latest/RData/>²¹

Usage

```
loadDaily()
```

Value

A *ws_monitor* object with PM2.5 monitoring data.

See Also

```
loadLatest
```

Examples

```
## Not run:
ca <- loadDaily() %>% monitor_subset(stateCodes='CA')

## End(Not run)
```

```
loadLatest
```

```
Load Recent PM2.5 Monitoring Data
```

Description

Wrapper function to load and combine the most recent data from AirNow, AIRSIS and WRCC:

```
airnow <- airnow_loadLatest()
airsis <- aairsis_loadLatest()
wrcc <- wrcc_loadLatest()
ws_monitor <- monitor_combine(list(airnow, aairsis, wrcc))
```

Avaiable RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/latest/RData/>²²

Usage

```
loadLatest()
```

Value

A *ws_monitor* object with PM2.5 monitoring data.

²¹<https://haze.airfire.org/monitoring/latest/RData/>

²²<https://haze.airfire.org/monitoring/latest/RData/>

See Also

airsis_loadDaily

Examples

```
## Not run:
ca <- loadLatest() %>% monitor_subset(stateCodes='CA')

## End(Not run)
```

monitor_aqi	<i>Calculate hourly NowCast-based AQI values</i>
-------------	--

Description

Nowcast and AQI algorithms are applied to the data in the ws_monitor object.

Usage

```
monitor_aqi(ws_monitor, aqiParameter = "pm25", nowcastVersion = "pm",
  includeShortTerm = FALSE)
```

Arguments

ws_monitor *ws_monitor* object

aqiParameter parameter type; used to define reference breakpointsTable

nowcastVersion
character identity specifying the type of nowcast algorithm to be used. See
?monitor_nowcast for more information.

includeShortTerm
calculate preliminary values starting with the 2nd hour

References

<https://docs.airnowapi.org/aq101>

https://www.ecfr.gov/cgi-bin/retrieveECFR?n=40y6.0.1.1.6#ap40.6.58_161.g

Examples

```
## Not run:
ws_monitor <- monitor_subset(Northwest_Megafires, tlim=c(20150815,20150831))
aqi <- monitor_aqi(ws_monitor)
monitor_timeseriesPlot(aqi, monitorID=aqi$meta$monitorID[1], ylab="PM25 AQI")

## End(Not run)
```

```
monitor_asDataframe
```

Return Monitor Data in a Single Dataframe

Description

Creates a dataframe with data from a `ws_monitor` object, essentially *flattening* the object. This is especially useful when monitoring data will be shared with non-R users working with spreadsheets. The returned dataframe will contain data from the monitor specified with `monitorID`.

The number of data columns in the returned dataframe can include all metadata as well as additional calculated values.

By default, the following, core columns are included in the dataframe:

- `utcTime` UTC datetime
- `localTime` local datetime
- `pm25` PM2.5 values in ug/m3

Any column from `ws_monitor$meta` may be included in the vector of `metaColumns`.

The following additional columns of data may be included by adding one of the following to the vector of `extraColumns{}`

- `aqi` hourly AQI values as calculated with `monitor_aqi()`
- `nowcast` hourly Nowcast values as calculated with `monitor_nowcast()`
- `dailyAvg` daily average PM2.5 values as calculated with `monitor_dailyStatistic()`

Usage

```
monitor_asDataframe(ws_monitor, monitorID = NULL, extraColumns = NULL,
  metaColumns = NULL, tlim = NULL)
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object
<code>monitorID</code>	monitor ID of interest (not needed if <code>ws_monitor</code> contains only one monitor)
<code>extraColumns</code>	optional vector of additional data columns to generate
<code>metaColumns</code>	optional vector of column names from <code>ws_monitor\$meta</code>
<code>tlim</code>	optional vector with start and end times (integer or character representing YYYYMM-MDD[HH] or POSIXct)

Value

A dataframe version of a `ws_monitor` object.

Note

The `tlim` argument is interpreted as localtime, not UTC.

See Also

```
monitor_aqi
monitor_nowcast
monitor_dailyStatistic
```

Examples

```
## Not run:
wa <- monitor_subset(Northwest_Megafires, stateCodes='WA')
Omak_df <- monitor_asDataframe(wa, monitorID='530470013_01',
                              extraColumns=c('nowcast', 'dailyAvg'),
                              metaColumns=c('aq5ID', 'siteName', 'timezone'),
                              tlim=c(20150801, 20150901))

## End(Not run)
```

monitor_collapse	<i>Collapse a ws_monitor Object into a ws_monitor Object with a Single Monitor</i>
------------------	--

Description

Collapses data from all the monitors in `ws_monitor` into a single-monitor `ws_monitor` object using the function provided in the `FUN` argument. The single-monitor result will be located at the mean longitude and latitude unless `longitude` and `latitude` parameters are specified.

Any columns of meta that are common to all monitors will be retained in the returned `ws_monitor` meta.

Usage

```
monitor_collapse(ws_monitor, longitude = NULL, latitude = NULL,
                 monitorID = "generated_id", FUN = mean, na.rm = TRUE, ...)
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object.
<code>longitude</code>	Longitude of the collapsed monitoring station.
<code>latitude</code>	Latitude of the collapsed monitoring station.
<code>monitorID</code>	Monitor ID assigned to the collapsed monitoring station.
<code>FUN</code>	Function to be applied to all the monitors at a single time index.
<code>na.rm</code>	Logical specifying whether NA values should be ignored when <code>FUN</code> is applied.
<code>...</code>	additional arguments to be passed on to the <code>apply()</code> function.

Value

A `ws_monitor` object with meta and data that for the the collapsed single monitor

Note

After FUN is applied, values of +Inf and -Inf are converted to NA. This is a convenience for the common case where FUN=min or FUN=max and some of the timesteps have all missing values. See the R documentation for min for an explanation.

Examples

```
N_M <- Northwest_Megafires
# monitor_leaflet(N_M) # to identify Spokane monitorIDs
Spokane <- monitor_subsetBy(N_M, stringr::str_detect(N_M$meta$monitorID, '^53063'))
Spokane_min <- monitor_collapse(Spokane, monitorID='Spokane_min', FUN=min)
Spokane_max <- monitor_collapse(Spokane, monitorID='Spokane_max', FUN=max)
monitor_timeseriesPlot(Spokane, tlim=c(20150619,20150626),
  style='gnats', shadedNight=TRUE)
monitor_timeseriesPlot(Spokane_max, col='red', type='s', add=TRUE)
monitor_timeseriesPlot(Spokane_min, col='blue', type='s', add=TRUE)
title('Spokane Range of PM2.5 Values, June 2015')
```

monitor_combine *Combine List of ws_monitor Objects into Single ws_monitor Object*

Description

Combines a list of one or more *ws_monitor* objects into a single *ws_monitor* object by merging the meta and data dataframes from each object in `monitorList`.

When `monitorList` contains only two *ws_monitor* objects the `monitor_combine()` function can be used to extend time ranges for monitorIDs that are found in both *ws_monitor* objects. This can be used to 'grow' a *ws_monitor* object by appending subsequent months or years. (Note, however, that this can be CPU intensive process.)

Usage

```
monitor_combine(monitorList)
```

Arguments

`monitorList` list containing one or more *ws_monitor* objects

Value

A *ws_monitor* object combining all monitoring data from `monitorList`.

Examples

```
## Not run:
monitorList <- list()
monitorList[[1]] <- airsis_createMonitorObject(20160701, 20161231, 'USFS', '1031')
monitorList[[2]] <- airsis_createMonitorObject(20160701, 20161231, 'USFS', '1032')
monitorList[[3]] <- airsis_createMonitorObject(20160701, 20161231, 'USFS', '1033')
monitorList[[4]] <- airsis_createMonitorObject(20160701, 20161231, 'USFS', '1034')
ws_monitor <- monitor_combine(monitorList)
monitor_leaflet(ws_monitor)

## End(Not run)
```

```
monitor_dailyBarplot
```

Create Daily Barplot

Description

Creates a bar plot showing daily average PM 2.5 values for a specific monitor in a *ws_monitor* object. Each bar is colored according to its AQI category.

This function is a wrapper around `base::barplot` and any arguments to that function may be used.

Each 'day' is the midnight-to-midnight period in the monitor local timezone. When `tlim` is used, it is converted to the monitor local timezone.

Usage

```
monitor_dailyBarplot(ws_monitor, monitorID = NULL, tlim = NULL,
  minHours = 18, gridPos = "", gridCol = "black", gridLwd = 0.5,
  gridLty = "solid", labels_x_nudge = 0, labels_y_nudge = 0, ...)
```

Arguments

<code>ws_monitor</code>	<i>ws_monitor</i> object
<code>monitorID</code>	monitor ID for a specific monitor in <i>ws_monitor</i> (optional if <i>ws_monitor</i> only has one monitor)
<code>tlim</code>	optional vector with start and end times (integer or character representing YYYYM-MDD[HH] or POSIXct)
<code>minHours</code>	minimum number of valid data hours required to calculate each daily average
<code>gridPos</code>	position of grid lines either 'over', 'under' ("" for no grid lines)
<code>gridCol</code>	color of grid lines (see graphical parameter 'col')
<code>gridLwd</code>	line width of grid lines (see graphical parameter 'lwd')
<code>gridLty</code>	type of grid lines (see graphical parameter 'lty')
<code>labels_x_nudge</code>	nudge x labels to the left

```

labels_y_nudge
                nudge y labels down
...            additional arguments to be passed to barplot()

```

Details

The `labels_x_nudge` and `labels_y_nudge` can be used to tweak the date labeling. Units used are the same as those in the plot.

Examples

```

## Not run:
N_M <- monitor_subset(Northwest_Megafires, tlim = c(20150715, 20150930))
main <- "Daily Average PM2.5 for Omak, WA"
monitor_dailyBarplot(N_M, monitorID = "530470013_01", main = main,
                    labels_x_nudge = 1)
addAQILegend(fill = rev(AQI$colors), pch = NULL)

## End(Not run)

```

```

monitor_dailyStatistic
                Calculate daily statistics

```

Description

Calculates daily statistics for each monitor in `ws_monitor`.

Usage

```

monitor_dailyStatistic(ws_monitor, FUN = get("mean"),
                      dayStart = "midnight", na.rm = TRUE, minHours = 18)

```

Arguments

<code>ws_monitor</code>	<i>ws_monitor</i> object
<code>FUN</code>	function used to collapse a day's worth of data into a single number for each monitor in the <code>ws_monitor</code> object
<code>dayStart</code>	one of <code>sunset</code> <code>midnight</code> <code>sunrise</code>
<code>na.rm</code>	logical value indicating whether NA values should be ignored
<code>minHours</code>	minimum number of valid data hours required to calculate each daily statistic

Details

Sunrise and sunset times are calculated based on the first monitor encountered. This should be accurate enough for all use cases involving co-located monitors. Monitors from different regions should have daily statistics calculated separately.

Value

A *ws_monitor* object with daily statistics for the local timezone.

Note

Note that the incoming *ws_monitor* object should have UTC (GMT) times and that this function calculates daily statistics based on local (clock) time. If you choose a date range based on UTC times this may result in an insufficient number of hours in the first and last daily records of the returned *ws_monitor* object.

The returned *ws_monitor* object has a daily time axis where each *datetime* is set to the beginning of each day, 00:00:00, local time.

Examples

```
## Not run:
N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150801,20150831))
WinthropID <- '530470010_01'
TwispID <- '530470009_01'
MethowValley <- monitor_subset(N_M,
                               tlim=c(20150801,20150831),
                               monitorIDs=c(WinthropID,TwispID))
MethowValley_dailyMean <- monitor_dailyStatistic(MethowValley,
                                                FUN=get('mean'),
                                                dayStart='midnight')

# Get the full Y scale
monitor_timeseriesPlot(MethowValley, style='gnats', col='transparent')
monitor_timeseriesPlot(MethowValley, monitorID=TwispID,
                       style='gnats', col='forestgreen', add=TRUE)
monitor_timeseriesPlot(MethowValley, monitorID=WinthropID,
                       style='gnats', col='purple', add=TRUE)
monitor_timeseriesPlot(MethowValley_dailyMean, monitorID=TwispID,
                       type='s', lwd=2, col='forestgreen', add=TRUE)
monitor_timeseriesPlot(MethowValley_dailyMean, monitorID=WinthropID,
                       type='s', lwd=2, col='purple', add=TRUE)

addAQILines()
addAQILegend("topleft", lwd=1, pch=NULL)
title("Winthrop & Twisp, Washington Daily Mean PM2.5, 2015")

## End(Not run)
```

```
monitor_dailyStatisticList
```

Calculate Daily Statistics

Description

Calculates daily statistics for each monitor in *ws_monitor*.

Usage

```
monitor_dailyStatisticList(ws_monitor, FUN = get("mean"),
  dayStart = "midnight", na.rm = TRUE, minHours = 18)
```

Arguments

<code>ws_monitor</code>	<i>ws_monitor</i> object
<code>FUN</code>	function used to collapse a day's worth of data into a single number for each monitor in the <i>ws_monitor</i> object
<code>dayStart</code>	one of <code>sunset</code> <code>midnight</code> <code>sunrise</code>
<code>na.rm</code>	logical value indicating whether NA values should be ignored
<code>minHours</code>	minimum number of valid data hours required to calculate each daily statistic

Details

Splits the *ws_monitor* object by `timezone` and applies the `monitor_dailyStatistic()` function separately for each timezone. See `monitor_dailyStatistic` for more details.

The results are returned as a list of *ws_monitor* objects with each element of the list named with the associated timezone. Note that each `ws_monitor$data$datetime` will be in local time. This is desirable as it ensures proper date formatting in tables and plots.

You should not attempt to reassemble a single *ws_monitor* object from the elements in this list.

Value

A list of *ws_monitor* objects with daily statistics for each local timezone.

References

`monitor_dailyStatistic`

Examples

```
## Not run:
airnow <- airnow_loadLatest()
nw <- monitor_subset(airnow, stateCodes = c('WA', 'OR', 'ID', 'MT'))
dailyList <- monitor_dailyStatisticList(nw)
monitor_leaflet(dailyList[["America/Los_Angeles"]])
monitor_leaflet(dailyList[["America/Boise"]])
monitor_leaflet(dailyList[["America/Denver"]])

## End(Not run)
```

`monitor_dailyThreshold`*Calculate Daily Counts of Values At or Above a Threshold*

Description

Calculates the number of hours per day each monitor in `ws_monitor` was at or above a given threshold

Usage

```
monitor_dailyThreshold(ws_monitor, threshold = "unhealthy",
  dayStart = "midnight", minHours = 0, na.rm = TRUE)
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object
<code>threshold</code>	AQI level name (e.g. "unhealthy") or numerical threshold at or above which a measurement is counted
<code>dayStart</code>	one of "sunset midnight sunrise"
<code>minHours</code>	minimum number of hourly observations required
<code>na.rm</code>	logical value indicating whether NA values should be ignored

Details

NOTE: The returned counts include values at OR ABOVE the given threshold; this applies to both categories and values. For example, passing a `threshold` argument = "unhealthy" will return a daily count of values that are unhealthy, very unhealthy, or extreme (i.e. ≥ 55.5), as will passing a `threshold` argument = 55.5.

AQI levels for `threshold` argument = one of "good|moderate|usg|unhealthy|very unhealthy|extreme"

Sunrise and sunset times are calculated based on the first monitor encountered. This should be accurate enough for all use cases involving co-located monitors. Monitors from different regions should have daily statistics calculated separately.

The returned `ws_monitor` object has a daily time axis where each time is set to 00:00, local time.

Value

A `ws_monitor` object with a daily count of hours at or above `threshold`.

Examples

```
## Not run:
N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150801,20150831))
Twisp <- monitor_subset(N_M, monitorIDs='530470009_01')
Twisp_daily <- monitor_dailyThreshold(Twisp, "unhealthy", dayStart='midnight', minHours=1)
monitor_timeseriesPlot(Twisp_daily, type='h', lwd=6, ylab="Hours")
title("Twisp, Washington Hours per day Above 'Unhealthy', 2015")

## End(Not run)
```

monitor_distance *Calculate distances from monitors to a location of interest*

Description

This function returns the distances (km) between monitoring sites and a location of interest. These distances can be used to create a mask identifying monitors within a certain radius of the location of interest.

Usage

```
monitor_distance(ws_monitor, longitude, latitude)
```

Arguments

ws_monitor	ws_monitor object
longitude	longitude of the location of interest
latitude	latitude of the location of interest

Value

Vector of of distances (km).

See Also

distance

Examples

```
N_M <- Northwest_Megafires
# Walla Walla
WW_lon <- -118.330278
WW_lat <- 46.065
distance <- monitor_distance(N_M, WW_lon, WW_lat)
closestIndex <- which(distance == min(distance))
distance[closestIndex]
N_M$meta[closestIndex,]
```

 monitor_downloadAnnual

Download annual PM2.5 monitoring data

Description

Downloads 'annual' data files into `dataDir` for later use. Downloaded versions of PWFSL monitoring .RData files allow users to work with the package without access to the internet. Once data are downloaded to `dataDir`, any of the data loading functions can be called with the `dataDir` argument to replace internet downloads with local file access.

The recommended directory for PWFSL monitoring data is "`~/data/monitoring/RData`".

For data during the last 45 days, use `monitor_downloadDaily()`.

For the most recent data, use `monitor_downloadLatest()`.

Currently supported parameters include the following:

1. PM2.5

Avaiable RData files can be seen at: <https://haze.airfire.org/monitoring/latest/RData/>²³

Usage

```
monitor_downloadAnnual(year = NULL, parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring",
  dataDir = "~/Data/monitoring/RData", ...)
```

Arguments

<code>year</code>	Desired year (integer or character representing YYYY).
<code>parameter</code>	Parameter of interest.
<code>baseUrl</code>	Base URL for data files.
<code>dataDir</code>	Local directory in which to save the data file.
<code>...</code>	Additional arguments passed to <code>download.file</code> .

See Also

`monitor_loadDaily`

Examples

```
## Not run:
monitor_loadLatest() %>%
  monitor_subset(stateCodes=CONUS) %>%
  monitor_map()

## End(Not run)
```

²³<https://haze.airfire.org/monitoring/latest/RData/>

```
monitor_downloadDaily
```

Download recent PM2.5 monitoring data

Description

Downloads 'daily' data files into `dataDir` for later use. Downloaded versions of PWFSL monitoring .RData files allow users to work with the package without access to the internet. Once data are downloaded to `dataDir`, any of the data loading functions can be called with the `dataDir` argument to replace internet downloads with local file access.

The recommended directory for PWFSL monitoring data is "`~/data/monitoring/RData`".

For the most recent data, use `monitor_downloadLatest()`.

For data extended more than 45 days into the past, use `monitor_downloadAnnual()`.

Currently supported parameters include the following:

1. PM2.5

Avaiable RData files can be seen at: <https://haze.airfire.org/monitoring/latest/RData/>²⁴

Usage

```
monitor_downloadDaily(parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData/",
  dataDir = "~/Data/monitoring/RData", ...)
```

Arguments

<code>parameter</code>	Parameter of interest.
<code>baseUrl</code>	Base URL for data files.
<code>dataDir</code>	Local directory in which to save the data file.
<code>...</code>	Additional arguments passed to <code>download.file</code> .

See Also

```
monitor_loadDaily
```

Examples

```
## Not run:
monitor_loadLatest() %>%
  monitor_subset(stateCodes=CONUS) %>%
  monitor_map()

## End(Not run)
```

²⁴<https://haze.airfire.org/monitoring/latest/RData/>

 monitor_downloadLatest

Download recent PM2.5 monitoring data

Description

Downloads 'latest' data files into `dataDir` for later use. Downloaded versions of PWFSL monitoring .RData files allow users to work with the package without access to the internet. Once data are downloaded to `dataDir`, any of the data loading functions can be called with the `dataDir` argument to replace internet downloads with local file access.

The recommended directory for PWFSL monitoring data is "`~/data/monitoring/RData`".

For daily updates covering the most recent 45 days, use `monitor_downloadDaily()`.

For data extended more than 45 days into the past, use `monitor_downloadAnnual()`.

Currently supported parameters include the following:

1. PM2.5

Avaiable RData files can be seen at: <https://haze.airfire.org/monitoring/latest/RData/>²⁵

Usage

```
monitor_downloadLatest(parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData/",
  dataDir = "~/Data/monitoring/RData", ...)
```

Arguments

<code>parameter</code>	Parameter of interest.
<code>baseUrl</code>	Base URL for data files.
<code>dataDir</code>	Local directory in which to save the data file.
<code>...</code>	Additional arguments passed to <code>download.file</code> .

See Also

`monitor_loadDaily`

Examples

```
## Not run:
monitor_loadLatest() %>%
  monitor_subset(stateCodes=CONUS) %>%
  monitor_map()

## End(Not run)
```

²⁵<https://haze.airfire.org/monitoring/latest/RData/>

monitor_dygraph *Create Interactive Time Series Plot*

Description

This function creates interactive graphs that will be displayed in RStudio's 'Viewer' tab.

Usage

```
monitor_dygraph(ws_monitor, title = "title",
  ylab = "PM2.5 Concentration", tlim = NULL, rollPeriod = 1,
  showLegend = TRUE)
```

Arguments

ws_monitor	<i>ws_monitor</i> object
title	title text
ylab	title for the y axis
tlim	optional vector with start and end times (integer or character representing YYYYMM-MDD[HH])
rollPeriod	rolling mean to be applied to the data
showLegend	logical to toggle display of the legend

Value

Initiates the interactive dygraph plot in RStudio's 'Viewer' tab.

Examples

```
## Not run:
# Napa Fires -- October, 2017
ca <- airnow_load(2017) %>%
  monitor_subset(tlim=c(20171001,20171101), stateCodes='CA')
Vallejo <- monitor_subset(ca, monitorIDs='060950004_01')
Napa_Fires <- monitor_subsetByDistance(ca,
  longitude = Vallejo$meta$longitude,
  latitude = Vallejo$meta$latitude,
  radius = 50)
monitor_dygraph(Napa_Fires, title='Napa Fires in California, Oct. 2017')

## End(Not run)
```

 monitor_extractDataFrame

Extract dataframes from ws_monitor objects

Description

These functions are convenient wrappers for extracting the dataframes that comprise a *ws_monitor* object. These functions are designed to be useful when manipulating data in a pipe chain.

Below is a table showing equivalent operations for each function.

Function	Equivalent Operation
monitor_extractData(ws_monitor)	ws_monitor[["data"]]
monitor_extractMeta(ws_monitor)	ws_monitor[["meta"]]

Usage

```
monitor_extractData(ws_monitor)
```

```
monitor_extractMeta(ws_monitor)
```

Arguments

ws_monitor *ws_monitor* object to extract dataframe from.

Value

A dataframe from the given *ws_monitor* object

Examples

```
## Not run:
ws_monitor <- Northwest_Megafires

NMData <- ws_monitor %>%
  monitor_subset(
    stateCodes = "WA",
    tlim = c(20150801, 20150831)
  ) %>%
  extract_data()

monitor_subset(
  stateCodes = "WA",
  tlim = c(20150801, 20150831)
) %>%
  extract_meta()

## End(Not run)
```

```
monitor_getCurrentStatus
    Get current status of monitors
```

Description

This function augments the metadata from a *ws_monitor* object with summarized and aggregate data from the *ws_monitor* object.

Usage

```
monitor_getCurrentStatus(ws_monitor, endTime = NULL,
    monitorURLBase = "http://tools.airfire.org/monitoring/v4/#!/?monitors=")
```

Arguments

<code>ws_monitor</code>	<i>ws_monitor</i> object.
<code>endTime</code>	Time to which the status of the monitors will be current. By default, it is the most recent time in <i>ws_monitor</i> . This time can be given as a POSIXct time, or a string/numeric value in ymd format (eg. 20190301). This time converted to UTC.
<code>monitorURLBase</code>	A URL prefix pointing to where more information about a monitor can be found. By default, it points to the AirFire monitoring site.

Value

A table containing the current status information for all the monitors in *ws_monitor*.

"Last" and "Previous"

The goal of this function is to provide useful information about what happened recently with each monitor in the provided *ws_monitor* object. Monitors sometimes don't consistently report data, however, and it's not useful to have NA's reported when there is still valid data at other times. To address this, `monitor_getCurrentStatus` uses *last* and *previous* valid times. These are the time when a monitor most recently reported data, and the most recent time of valid data before that, respectively. By reporting on these times, the function ensures that valid data is returned and provides information on how outdated this information is.

Calculating latency

According to <https://docs.airnowapi.org/docs/HourlyDataFactSheet.pdf> a datum assigned to 2pm represents the average of data between 2pm and 3pm. So, if we check at 3:15pm and see that we have a value for 2pm but not 3pm then the data are completely up-to-date with zero latency.

`monitor_getCurrentStatus()` defines latency as the difference in time between the given time index and the next most recent time index. If there is no more recent time index, then the difference is measured to the given `endTime` parameter. These differences are recorded in hours.

For example, if the recorded values for a monitor are [16.2, 15.8, 16.4, NA, 14.0, 12.5, NA, NA, 13.3, NA], then the last valid time index is 9, and the previous valid time index is 6. The last latency is then 1 (hour), and the previous latency is 3 (hours).

Summary data

The table created by `monitor_getCurrentStatus()` includes summary information for the data part of the given `ws_monitor` object. The summaries included are listed below with a description:

yesterday_pm25_24hr Daily AQI value for the day prior to `endTime`
last_nowcast_1hr Last valid NowCast measurement
last_PM2.5_1hr Last valid raw PM2.5 measurement
last_PM2.5_3hr Mean of the last valid raw PM2.5 measurement with the preceding two measurements
previous_nowcast_1hr Previous valid NowCast measurement
previous_PM2.5_1hr Previous valid raw PM2.5 measurement
previous_PM2.5_3hr Mean of the previous valid raw PM2.5 measurement with the preceding two measurements

It should be noted that all averages are "right-aligned", meaning that the three hour mean of data at time n will comprise of the data at times $[n-2, n-1, n]$. Data for $n-2$ and $n-1$ is not guaranteed to exist, so a three hour average may include 1 to 3 data points.

Event flags

The table created by `monitor_getCurrentStatus()` also includes binary flags representing events that may have occurred for a monitor within the bounds of the specified end time and data in the `ws_monitor` object. Each flag is listed below with its corresponding meaning:

last_nowcastLevel NowCast level at the last valid time
previous_nowcastLevel NowCast level at the previous valid time
NR6 Monitor not reporting for more than 6 hours
NEW6 New monitor reporting in the last 6 hours
USG6 NowCast level increased to Unhealthy for Sensitive Groups in the last 6 hours
U6 NowCast level increased to Unhealthy in the last 6 hours
VU6 NowCast level increased to Very Unhealthy in the last 6 hours
HAZ6 NowCast level increased to Hazardous in the last 6 hours
MOD6 NowCast level decreased to Moderate or Good in the last 6 hours
MAL6 Monitor malfunctioning the last 6 hours (not currently implemented)

Examples

```
## Not run:
ws_monitor <- monitor_loadLatest() %>% monitor_subset(stateCodes = "WA")
statusTbl <- monitor_getCurrentStatus(ws_monitor)

## End(Not run)
```

```
monitor_getDailyMean
```

Calculate daily means for a ws_monitor object

Description

Calculates and returns daily means for a monitor. If either `startdate` or `enddate` is `NULL`, a single value is returned for that date.

Usage

```
monitor_getDailyMean(ws_monitor, monitorID = NULL, startdate = NULL,
  enddate = NULL)
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object
<code>monitorID</code>	monitor ID of interest
<code>startdate</code>	desired start date (integer or character in Ymd format or POSIXct)
<code>enddate</code>	desired end date (integer or character in Ymd format or POSIXct)

Value

A dataframe of daily means.

Examples

```
monitor_getDailyMean(PWFSLSmoke::Carmel_Valley,
  startdate = "2016-08-01",
  enddate = "2016-08-08")
```

```
monitor_hourlyBarplot
```

Create Hourly Barplot

Description

Creates a bar plot showing hourly PM 2.5 values for a specific monitor in a `ws_monitor` object. Colors are assigned to one of the following styles:

- `AQI` – hourly values colored with AQI colors using AQI 24-hour breaks
- `brownScaleAQI` – hourly values colored with brownscale colors using AQI 24-hour breaks
- `grayScaleAQI` – hourly values colored grayscale colors using AQI 24-hour breaks

Usage

```
monitor_hourlyBarplot(ws_monitor, monitorID = NULL, tlim = NULL,
  localTime = TRUE, style = "AQI", shadedNight = TRUE,
  gridPos = "", gridCol = "black", gridLwd = 0.5,
  gridLty = "solid", labels_x_nudge = 0, labels_y_nudge = 0,
  dayCol = "black", dayLwd = 2, dayLty = "solid",
  hourCol = "black", hourLwd = 1, hourLty = "solid",
  hourInterval = 6, ...)
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object
<code>monitorID</code>	monitor ID for a specific monitor in <code>ws_monitor</code> (optional if <code>ws_monitor</code> only has one monitor)
<code>tlim</code>	optional vector with start and end times (integer or character representing YYYYMM-MDD[HH])
<code>localTime</code>	logical specifying whether <code>tlim</code> is in local time or UTC
<code>style</code>	named style specification ('AirFire')
<code>shadedNight</code>	add nighttime shading
<code>gridPos</code>	position of grid lines either 'over', 'under' ("" for no grid lines)
<code>gridCol</code>	grid color
<code>gridLwd</code>	grid line width
<code>gridLty</code>	grid line type
<code>labels_x_nudge</code>	nudge x labels to the left
<code>labels_y_nudge</code>	nudge y labels down
<code>dayCol</code>	day boundary color
<code>dayLwd</code>	day boundary line width (set to 0 to omit day lines)
<code>dayLty</code>	day boundary type
<code>hourCol</code>	hour boundary color
<code>hourLwd</code>	hour boundary line width (set to 0 to omit hour lines)
<code>hourLty</code>	hour boundary type
<code>hourInterval</code>	interval for hour boundary lines
<code>...</code>	additional arguments to be passed to <code>barplot()</code>

Details

The `labels_x_nudge` and `labels_y_nudge` can be used to tweak the date labeling. Units used are the same as those in the plot.

Examples

```
C_V <- monitor_subset(Carmel_Valley, tlim = c(2016080800,2016081023),
                     timezone = "America/Los_Angeles")
monitor_hourlyBarplot(C_V, main = "1-Hourly Average PM2.5",
                     labels_x_nudge = 1, labels_y_nudge = 0)
```

monitor_isEmpty *Test for an Empty ws_monitor Object*

Description

Convenience function for `nrow(ws_monitor$meta) == 0`. This makes for more readable code in the many functions that need to test for this.

Usage

```
monitor_isEmpty(ws_monitor)
```

Arguments

`ws_monitor` *ws_monitor* object

Value

TRUE if no monitors exist in `ws_monitor`, FALSE otherwise.

Examples

```
monitor_isEmpty(Carmel_Valley)
```

monitor_isMonitor *Test for an correct structure of ws_monitor Object*

Description

The `ws_monitor` is checked for the 'ws_monitor' class name and presence of core metadata columns:

- monitorID – per deployment unique ID
- longitude – decimal degrees E
- latitude – decimal degrees N
- elevation – height above sea level in meters
- timezone – olson timezone
- countryCode – ISO 3166-1 alpha-2
- stateCode – ISO 3166-2 alpha-2

Usage

```
monitor_isMonitor(ws_monitor)
```

Arguments

ws_monitor *ws_monitor* object

Value

TRUE if *ws_monitor* has the correct structure, FALSE otherwise.

Examples

```
monitor_isEmpty(Carmel_Valley)
```

monitor_isolate *Isolate Individual Monitors*

Description

Filters *ws_monitor* according to the parameters passed in. If any parameter is not specified, that parameter will not be used in the filtering.

After filtering, each *monitorID* found in *ws_monitor* is extracted and its data dataframe is restricted to the times from when that monitor first datapoint until its last datapoint.

This function is useful when *ws_monitor* objects are created for mobile monitors that are deployed to different locations in different years.

Usage

```
monitor_isolate(ws_monitor, xlim = NULL, ylim = NULL, tlim = NULL,
  monitorIDs = NULL, stateCodes = NULL, timezone = "UTC")
```

Arguments

ws_monitor *ws_monitor* object

xlim optional vector with low and high longitude limits

ylim optional vector with low and high latitude limits

tlim optional vector with start and end times (integer or character representing YYYYMM-MDD[HH] or POSIXct)

monitorIDs optional vector of *monitorIDs*

stateCodes optional vector of *stateCodes*

timezone Olson timezone passed to `link{parseDatetime}` when parsing numeric *tlim*

Value

A list of isolated ws_monitor objects.

See Also

monitor_subset

Examples

```
N_M <- Northwest_Megafires
# monitor_leaflet(N_M) # to identify Spokane monitorIDs
Spokane <- monitor_subsetBy(N_M, stringr::str_detect(N_M$meta$monitorID, '^53063'))
Spokane$meta$monitorID
monitorList <- monitor_isolate(Spokane)
names(monitorList)
```

monitor_isTidy	<i>Check if data is tidy-formatted ws_monitor data</i>
----------------	--

Description

Verifies that the given data can be treated as tidy-formatted "ws_monitor" data. This is done by verifying that the data is a tibble data.frame object with columns for information in all 'ws_monitor' objects.

Usage

```
monitor_isTidy(data = NULL)
```

Arguments

data Data to validate.

Value

True if the data is in a recognized 'Tidy' format, otherwise False.

Examples

```
ws_monitor <- monitor_subset(
  Northwest_Megafires,
  monitorIDs = c('530470009_01', '530470010_01')
)

ws_monTidy <- monitor_toTidy(ws_monitor)
monitor_isTidy(ws_monTidy)

## Not run:
monitor_isTidy(ws_monitor)
```

```
## End(Not run)
```

monitor_join	<i>Merge Data for Monitors with Shared monitorIDs</i>
--------------	---

Description

For each monitor in `monitorIDs`, an attempt is made to merge the associated data from `ws_monitor1` and `ws_monitor2` and.

This is useful when the same `monitorID` appears in different `ws_monitor` objects representing different time periods. The returned `ws_monitor` object will cover both time periods.

Usage

```
monitor_join(ws_monitor1 = NULL, ws_monitor2 = NULL,
             monitorIDs = NULL)
```

Arguments

<code>ws_monitor1</code>	<i>ws_monitor</i> object
<code>ws_monitor2</code>	<i>ws_monitor</i> object
<code>monitorIDs</code>	vector of shared <code>monitorIDs</code> that are to be joined together. Defaults to all shared <code>monitorIDs</code> .

Value

A *ws_monitor* object with merged timeseries.

Examples

```
## Not run:
Jul <- monitor_subset(Northwest_Megafires,
                     tlim=c(2015070100,2015073123),
                     timezone='America/Los_Angeles')
Aug <- monitor_subset(Northwest_Megafires,
                     tlim=c(2015080100,2015083123),
                     timezone='America/Los_Angeles')
Methow_Valley <- monitor_join(Jul, Aug, monitorIDs=c('530470010_01','530470009_01'))

## End(Not run)
```

monitor_leaflet *Leaflet interactive map of monitoring stations*

Description

This function creates interactive maps that will be displayed in RStudio's 'Viewer' tab. The `slice` argument is used to collapse a `ws_monitor` timeseries into a single value. If `slice` is an integer, that row index will be selected from the `ws_monitor$data` dataframe. If `slice` is a function (unquoted), that function will be applied to the timeseries with the argument `na.rm=TRUE` (e.g. `max(..., na.rm=TRUE)`).

If `slice` is a user defined function it will be used with argument `na.rm=TRUE` to collapse the time dimension. Thus, user defined functions must accept `na.rm` as an argument.

Usage

```
monitor_leaflet(ws_monitor, slice = get("max"), breaks = AQI$breaks_24,
  colors = AQI$colors, labels = AQI$names,
  legendTitle = "Max AQI Level", radius = 10, opacity = 0.7,
  maptype = "terrain", popupInfo = c("siteName", "monitorID",
  "elevation"))
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object
<code>slice</code>	either a time index or a function used to collapse the time axis – defaults to <code>get('max')</code>
<code>breaks</code>	set of breaks used to assign colors
<code>colors</code>	a set of colors for different levels of air quality data determined by <code>breaks</code>
<code>labels</code>	a set of text labels, one for each color
<code>legendTitle</code>	legend title
<code>radius</code>	radius of monitor circles
<code>opacity</code>	opacity of monitor circles
<code>maptype</code>	optional name of leaflet ProviderTiles to use, e.g. "terrain"
<code>popupInfo</code>	a vector of column names from <code>ws_monitor\$meta</code> to be shown in a popup window

Details

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

1. "roadmap" – "OpenStreetMap"
2. "satellite" – "Esri.WorldImagery"
3. "terrain" – "Esri.WorldTopoMap"

4. "toner" – "Stamen.Toner"

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

Value

Invisibly returns a leaflet map of class "leaflet".

Examples

```
## Not run:
# Napa Fires -- October, 2017
ca <- airnow_load(2017) %>%
  monitor_subset(tlim = c(20171001,20171101), stateCodes = 'CA')
v_low <- AQI$breaks_24[5]
CA_very_unhealthy_monitors <- monitor_subset(ca, vlim = c(v_low, Inf))
monitor_leaflet(CA_very_unhealthy_monitors,
  legendTitle = "October, 2017",
  maptype = "toner")

## End(Not run)
```

monitor_load	<i>Load PM2.5 monitoring data</i>
--------------	-----------------------------------

Description

Loads monitoring data for a given time range. Data from AirNow, AIRSIS and WRCC are combined into a single *ws_monitor* object.

Archival datasets are joined with 'daily' and 'latest' datasets as needed to satisfy the requested date range.

Usage

```
monitor_load(startdate = NULL, enddate = NULL, monitorIDs = NULL,
  parameter = "PM2.5", baseUrl = "https://haze.airfire.org/monitoring",
  dataDir = NULL, aqsPreference = "airnow")
```

Arguments

startdate	Desired start date (integer or character in ymd[hms] format or POSIXct).
enddate	Desired end date (integer or character in ymd[hms] format or POSIXct).
monitorIDs	Optional vector of monitorIDs.
parameter	Parameter of interest.
baseUrl	Base URL for data files.

dataDir Local directory containing monitoring data files.
 aqsPreference Preferred data source for AQS data when annual data files are available from both 'epa' and 'airnow'.

Value

A *ws_monitor* object with PM2.5 monitoring data.

Note

Joining datasets is a computationally expensive task when many monitors are involved. It is highly recommend that `monitorIDs` be specified when loading recent data with this function.

See Also

loadDaily
 loadLatest

Examples

```
## Not run:
ca <- monitor_load(20170601,20171001) %>% monitor_subset(stateCodes='CA')

## End(Not run)
```

monitor_loadAnnual *Load annual PM2.5 monitoring data*

Description

Wrapper function to load and combine annual data from AirNow, AIRSIS and WRCC.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The annual files loaded by this function are updated on the 15th of each month and cover the period from the beginning of the year to the end of the last month.

For data during the last 45 days, use `monitor_loadDaily()`.

For the most recent data, use `monitor_loadLatest()`.

Currently supported parameters include the following:

1. PM2.5

Avaiable RData files can be seen at: <https://haze.airfire.org/monitoring/>²⁶

²⁶<https://haze.airfire.org/monitoring/>

Usage

```
monitor_loadAnnual(year = NULL, parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring", dataDir = NULL,
  aqsPreference = "airnow")
```

Arguments

year	Desired year (integer or character representing YYYY).
parameter	Parameter of interest.
baseUrl	Base URL for data files.
dataDir	Local directory containing 'daily' data files.
aqsPreference	Preferred data source for AQS data when annual data files are available from both 'epa' and 'airnow'.

Value

A *ws_monitor* object with PM2.5 monitoring data.

See Also

```
monitor_loadDaily
monitor_loadLatest
```

Examples

```
## Not run:
monitor_loadAnnual(2014) %>%
  monitor_subset(stateCodes='MT', tlim=c(20140801,20140901)) %>%
  monitor_map()

## End(Not run)
```

monitor_loadDaily *Load recent PM2.5 monitoring data*

Description

Wrapper function to load and combine recent data from AirNow, AIRSIS and WRCC:

```
airnow <- airnow_loadDaily()
airsis <- aairsis_loadDaily()
wrcc <- wrcc_loadDaily()
ws_monitor <- monitor_combine(list(airnow, aairsis, wrcc))
```

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The daily files loaded by this function are updated once a day, shortly after midnight and contain data for the previous 45 days.

For the most recent data, use `monitor_loadLatest()`.

For data extended more than 45 days into the past, use `monitor_load()`.

Currently supported parameters include the following:

1. PM2.5

Avaiable RData files can be seen at: <https://haze.airfire.org/monitoring/latest/RData>²⁷

Usage

```
monitor_loadDaily(parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL)
```

Arguments

<code>parameter</code>	Parameter of interest.
<code>baseUrl</code>	Base URL for 'daily' AirNow data files.
<code>dataDir</code>	Local directory containing 'daily' data files.

Value

A `ws_monitor` object with PM2.5 monitoring data.

See Also

```
monitor_load
monitor_loadLatest
monitor_loadAnnual
```

Examples

```
## Not run:
monitor_loadDaily() %>%
  monitor_subset(stateCodes=CONUS) %>%
  monitor_map()

## End(Not run)
```

²⁷<https://haze.airfire.org/monitoring/latest/RData/>

monitor_loadLatest *Load most recent PM2.5 monitoring data*

Description

Wrapper function to load and combine recent data from AirNow, AIRSIS and WRCC:

```
airnow <- airnow_loadLatest()
airsis <- aairsis_loadLatest()
wrcc <- wrcc_loadLatest()
ws_monitor <- monitor_combine(list(airnow, aairsis, wrcc))
```

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The files loaded by this function are updated multiple times an hour and contain data for the previous 10 days.

For daily updates covering the most recent 45 days, use `monitor_loadDaily()`.

For data extended more than 45 days into the past, use `monitor_load()`.

Currently supported parameters include the following:

1. PM2.5

Avaiable RData files can be seen at: <https://haze.airfire.org/monitoring/latest/RData/>²⁸

Usage

```
monitor_loadLatest(parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData/",
  dataDir = NULL)
```

Arguments

<code>parameter</code>	Parameter of interest.
<code>baseUrl</code>	Base URL for 'daily' AirNow data files.
<code>dataDir</code>	Local directory containing 'daily' data files.

Value

A `ws_monitor` object with PM2.5 monitoring data.

See Also

```
monitor_load
monitor_loadAnnual
monitor_loadDaily
```

²⁸<https://haze.airfire.org/monitoring/latest/RData/>

Examples

```
## Not run:
monitor_loadLatest() %>%
  monitor_subset(stateCodes=CONUS) %>%
  monitor_map()

## End(Not run)
```

 monitor_map

Static map of monitoring stations

Description

Creates a map of monitoring stations in a given `ws_monitor` object. Individual monitor timeseries are reduced to a single value by applying the function passed in as `slice` to the entire timeseries of each monitor with `na.rm=TRUE`. These values are then plotted over a map of the United States. Any additional arguments specified in `'...'` are passed on to the `points()` function.

If `slice` is an integer, it will be used as an index to pull out a single timestep.

If `slice` is a function (not a function name) it will be used with argument `na.rm=TRUE` to collapse the time dimension. Thus, any user defined functions passed in as `slice` must accept `na.rm` as a parameter.

Usage

```
monitor_map(ws_monitor, slice = get("max"), breaks = AQI$breaks_24,
  colors = AQI$colors, pch = par("pch"), cex = par("cex"),
  stateCol = "grey60", stateLwd = 2, countyCol = "grey70",
  countyLwd = 1, add = FALSE, ...)
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object
<code>slice</code>	either a time index or a function used to collapse the time axis
<code>breaks</code>	set of breaks used to assign colors
<code>colors</code>	set of colors must be one less than the number of breaks
<code>pch</code>	Plot symbols used to draw points on the map.
<code>cex</code>	the amount that the points will be magnified on the map
<code>stateCol</code>	color for state outlines on the map
<code>stateLwd</code>	width for state outlines
<code>countyCol</code>	color for county outline on the map
<code>countyLwd</code>	width for county outlines
<code>add</code>	logical specifying whether to add to the current plot
<code>...</code>	additional arguments passed to <code>maps::map()</code> such as <code>'projection'</code> or <code>'parameters'</code>

Details

Using a single number for the `breaks` argument will result in the use of quantiles to determine a set of breaks appropriate for the number of colors.

Examples

```
## Not run:
N_M <- monitor_subset(Northwest_Megafires, tlim = c(20150821,20150828))
monitor_map(N_M, cex = 2)
addAQILegend()

## End(Not run)
```

`monitor_nowcast` *Apply Nowcast Algorithm to ws_monitor Object*

Description

A Nowcast algorithm is applied to the data in in the `ws_monitor` object. The `version` argument specifies the minimum weight factor and number of hours to be considered in the calculation.

Available versions include:

1. `pm`: hours=12, weight=0.5
2. `pmAsian`: hours=3, weight=0.1
3. `ozone`: hours=8, weight=NA

The default, `version='pm'`, is appropriate for typical usage.

Usage

```
monitor_nowcast(ws_monitor, version = "pm", includeShortTerm = FALSE)
```

Arguments

`ws_monitor` *ws_monitor* object
`version` character identity specifying the type of nowcast algorithm to be used
`includeShortTerm`
 calculate preliminary NowCast values starting with the 2nd hour

Details

This function calculates the current hour's NowCast value based on the value for the given hour and the previous N-1 hours, where N is the number of hours corresponding to the `version` argument (see **Description** above). For example, if `version=pm`, then the NowCast value for Hour 12 is based on the data from Hours 1-12.

The function requires valid data for at least two of the three latest hours; NA's are returned for hours where this condition is not met.

By default, the function will not return a valid value until the Nth hour. If `includeShortTerm=TRUE`, the function will return a valid value after only the 2nd hour (provided, of course, that both hours are valid).

Calculated Nowcast values are truncated to the nearest .1 ug/m3 for 'pm' and nearest .001 ppm for 'ozone' regardless of the precision of the data in the incoming `ws_monitor` object.

Value

A `ws_monitor` object with data that have been processed by the Nowcast algorithm.

References

[https://en.wikipedia.org/wiki/Nowcast_\(Air_Quality_Index\)](https://en.wikipedia.org/wiki/Nowcast_(Air_Quality_Index))

https://www3.epa.gov/airnow/ani/pm25_aqi_reporting_nowcast_overview.pdf

<https://aqicn.org/faq/2015-03-15/air-quality-nowcast-a-beginners-guide/>

<https://airnow.zendesk.com/hc/en-us/articles/211625598-How-does-AirNow-make-the-Cu>

<https://forum.airnowtech.org/t/the-nowcast-for-ozone-and-pm/172>

<https://forum.airnowtech.org/t/the-aqi-equation/169>

<https://forum.airnowtech.org/t/how-does-airnow-handle-negative-hourly-concentrations/143>

Examples

```
## Not run:
N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150815,20150831))
Omak <- monitor_subset(N_M, monitorIDs='530470013_01')
Omak_nowcast <- monitor_nowcast(Omak, includeShortTerm=TRUE)
monitor_timeseriesPlot(Omak, type='l', lwd=2)
monitor_timeseriesPlot(Omak_nowcast, add=TRUE, type='l', col='purple', lwd=2)
addAQILines()
addAQILegend(lwd=1, pch=NULL)
legend("topleft", lwd=2, col=c('black','purple'), legend=c('hourly','nowcast'))
title("Omak, Washington Hourly and Nowcast PM2.5 Values in August, 2015")
# Zooming in to check on handling of missing values
monitor_timeseriesPlot(Omak, tlim=c(20150823,20150825))
monitor_timeseriesPlot(Omak_nowcast, tlim=c(20150823,20150825), pch=16,col='red',type='b', a
abline(v=Omak$data[is.na(Omak$data[,2]),1])
title("Missing values")

## End(Not run)
```

`monitor_performance`*Calculate Monitor Prediction Performance*

Description

This function uses *confusion matrix* analysis to calculate different measures of predictive performance for every timeseries found in `predicted` with respect to the observed values found in the single timeseries found in `observed`.

The requested metric is returned in a dataframe organized with one row per monitor, all available metrics are returned.

Usage

```
monitor_performance(predicted, observed, t1, t2, metric = NULL,
  FPCost = 1, FNCost = 1)
```

Arguments

<code>predicted</code>	<code>ws_monitor</code> object with predicted data
<code>observed</code>	<code>ws_monitor</code> object with observed data
<code>t1</code>	value used to classify predicted measurements
<code>t2</code>	threshold used to classify observed measurements
<code>metric</code>	<i>confusion matrix</i> metric to be used
<code>FPCost</code>	cost associated with false positives (type II error)
<code>FNCost</code>	cost associated with false negatives (type I error)

Value

Dataframe of monitors vs named measure of performance.

See Also

`monitor_performanceMap`
`skill_confusionMatrix`

Examples

```
## Not run:
# If daily avg data were the prediction and Spokane were
# the observed, which WA State monitors had skill?
wa <- airnow_loadAnnual(2017) %>% monitor_subset(stateCodes='WA')
wa_dailyAvg <- monitor_dailyStatistic(wa, mean)
Spokane_dailyAvg <- monitor_subset(wa_dailyAvg, monitorIDs='530630021_01')
threshold <- AQI$breaks_24[4] # Unhealthy
performanceMetrics <- monitor_performance(wa_dailyAvg,
```

```

                                Spokane_dailyAvg,
                                threshold, threshold)
monitorIDs <- rownames(performanceMetrics)
mask <- performanceMetrics$heidkeSkill &
      !is.na(performanceMetrics$heidkeSkill)
skillfulIDs <- monitorIDs[mask]
skillful <- monitor_subset(wa_dailyAvg, monitorIDs=skillfulIDs)
monitor_leaflet(skillful)

## End(Not run)

```

```
monitor_performanceMap
```

Create map of monitor prediction performance

Description

This function uses *confusion matrix* analysis to calculate different measures of predictive performance for every timeseries found in predicted with respect to the observed values found in the single timeseries found in observed.

Using a single number for the `breaks` argument will cause the algorithm to use quantiles to determine breaks.

Usage

```

monitor_performanceMap(predicted, observed, threshold = AQI$breaks_24[3],
  cex = par("cex"), sizeBy = NULL, colorBy = "heidikeSkill",
  breaks = c(-Inf, 0.5, 0.6, 0.7, 0.8, Inf),
  paletteFunc = grDevices::colorRampPalette(RColorBrewer::brewer.pal(length(breaks)
  "Purples")[-1]), showLegend = TRUE, legendPos = "topright",
  stateCol = "grey60", stateLwd = 2, countyCol = "grey70",
  countyLwd = 1, add = FALSE, ...)

```

Arguments

<code>predicted</code>	<code>ws_monitor</code> object with predicted values
<code>observed</code>	<code>ws_monitor</code> object with observed values
<code>threshold</code>	value used to classify predicted and observed measurements
<code>cex</code>	the amount that the points will be magnified on the map
<code>sizeBy</code>	name of the metric used to create relative sizing
<code>colorBy</code>	name of the metric used to create relative colors
<code>breaks</code>	set of breaks used to assign colors or a single integer used to provide quantile based breaks - Must also specify the <code>colorBy</code> paramater
<code>paletteFunc</code>	a palette generating function as returned by <code>colorRampPalette</code>
<code>showLegend</code>	logical specifying whether to add a legend (default: TRUE)

legendPos	legend position passed to legend()
stateCol	color for state outlines on the map
stateLwd	width for state outlines
countyCol	color for county outline on the map
countyLwd	width for county outlines
add	logical specifying whether to add to the current plot
...	additional arguments to be passed to the maps::map() function such as graphical parameters (see code?par)

Details

Setting either `sizeBy` or `colorBy` to `NULL` will cause the size/colors to remain constant.

See Also

`monitor_performance`

Examples

```
## Not run:
# Napa Fires -- October, 2017
ca <- airnow_load(2017) %>%
  monitor_subset(tlim=c(20171001,20171101), stateCodes='CA')
Vallejo <- monitor_subset(ca, monitorIDs='060950004_01')
Napa_Fires <- monitor_subsetByDistance(ca,
                                     longitude = Vallejo$meta$longitude,
                                     latitude = Vallejo$meta$latitude,
                                     radius = 50)

monitor_performanceMap(ca, Vallejo, cex=2)
title('Heidke Skill of monitors predicting another monitor.')

## End(Not run)
```

monitor_print

Print monitor data as CSV

Description

Prints out the contents of the `ws_monitor` object as **CSV**. By default, the output is a text string with "human readable" CSV that includes both `meta` and `data`. When saved as a file, this format is useful for point-and-click spreadsheet users who want to have everything on a single sheet.

To obtain machine parseable CSV strings you can use `metaOnly` or `dataOnly` which are mutually exclusive but which return CSV strings that can be automatically ingested.

By default, the CSV formatted text is printed to the console as well as returned invisibly but not saved to a file unless `saveFile` is specified.

Usage

```
monitor_print(ws_monitor, saveFile = NULL, metaOnly = FALSE,
             dataOnly = FALSE, quietly = FALSE)
```

Arguments

<code>ws_monitor</code>	<i>ws_monitor</i> object
<code>saveFile</code>	optional filename where CSV will be written
<code>metaOnly</code>	flag specifying whether to return <code>ws_monitor\$meta</code> only as a machine parseable CSV
<code>dataOnly</code>	flag specifying whether to return <code>ws_monitor\$data</code> only as a machine parseable CSV
<code>quietly</code>	do not print to console, just return the string representation of the CSV

Note

The `monitor_writeCSV` function is an alias for this function but defaults to `quietly = TRUE`.

Examples

```
data("Carmel_Valley")
Carmel_Valley <- monitor_subset(Carmel_Valley, tlim = c(20160802,20160803))
monitor_print(Carmel_Valley)
monitor_print(Carmel_Valley, metaOnly = TRUE)
monitor_print(Carmel_Valley, dataOnly = TRUE)
```

`monitor_reorder` *Reorder a `ws_monitor` object*

Description

This function is a convenience function that merely wraps the `monitor_subset` function which reorders as well as subsets.

Usage

```
monitor_reorder(ws_monitor, monitorIDs = NULL, dropMonitors = FALSE)
```

Arguments

<code>ws_monitor</code>	<i>ws_monitor</i> object
<code>monitorIDs</code>	Optional vector of monitor IDs used to reorder the meta and data dataframes.
<code>dropMonitors</code>	Logical specifying whether to remove monitors with no data.

Value

A *ws_monitor* object reordered to match `monitorIDs`.

`monitor_replaceData`*Replace ws_monitor Data with Another Value*

Description

Use an R expression to identify values for replacement.

The RR expression given in `filter` is used to identify elements in `ws_monitor$data` that should be replaced. Typical usage would include

1. replacing negative values with 0
2. replacing unreasonably high values with NA

Expressions should use `data` for the left hand side of the comparison.

Usage

```
monitor_replaceData(ws_monitor, filter, value)
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object
<code>filter</code>	an RR expression used to identify values for replacement
<code>value</code>	replacement value

Examples

```
wa <- monitor_subset(Northwest_Megafires, stateCodes='WA')
wa_zero <- monitor_replaceData(wa, data < 0, 0)
```

`monitor_rollingMean`*Calculate Rolling Means*

Description

Calculates rolling means for each monitor in `ws_monitor` using the `openair::rollingMean()` function

Usage

```
monitor_rollingMean(ws_monitor, width = 8, data.thresh = 75,
  align = "center")
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object
<code>width</code>	number of periods to average (e.g. for hourly data, <code>width = 24</code> calculates 24-hour rolling means)
<code>data.thresh</code>	minimum number of valid observations required as a percent of <code>width</code> ; NA is returned if insufficient valid data to calculate mean
<code>align</code>	alignment of averaging window relative to point being calculated; one of "left center right"

Details

- `align = 'left'`: Forward roll, using hour of interest and the $(width-1)$ subsequent hours (e.g. 3-hr left-aligned roll for Hr 5 will consist of average of Hrs 5, 6 and 7)
- `align = 'right'`: Backwards roll, using hour of interest and the $(width-1)$ prior hours (e.g. 3-hr right-aligned roll for Hr 5 will consist of average of Hrs 3, 4 and 5)
- `align = 'center'` for odd `width`: Average of hour of interest and $(width-1)/2$ on either side (e.g. 3-hr center-aligned roll for Hr 5 will consist of average of Hrs 4, 5 and 6)
- `align = 'center'` for even `width`: Average of hour of interest and $(width/2)-1$ hours prior and `width/2` hours after (e.g. 4-hr center-aligned roll for Hr 5 will consist of average of Hrs 4, 5, 6 and 7)

Value

A `ws_monitor` object with data that have been processed by a rolling mean algorithm.

Examples

```
N_M <- Northwest_Megafires
wa_smoky <- monitor_subset(N_M, stateCodes='WA', tlim=c(20150801, 20150808), vlim=c(100,Inf))
wa_smoky_3hr <- monitor_rollingMean(wa_smoky, width=3, align="center")
wa_smoky_24hr <- monitor_rollingMean(wa_smoky, width=24, align="right")
monitor_timeseriesPlot(wa_smoky, type='l', shadedNight=TRUE)
monitor_timeseriesPlot(wa_smoky_3hr, type='l', col='red', add=TRUE)
monitor_timeseriesPlot(wa_smoky_24hr, type='l', col='blue', lwd=2, add=TRUE)
legend('topright', c("hourly", "3-hourly", "24-hourly"),
      col=c('black', 'red', 'blue'), lwd=c(1,1,2))
title('Smoky Monitors in Washington -- August, 2015')
```

```
monitor_rollingMeanPlot
```

Create Rolling Mean Plot

Description

Creates a plot of individual (e.g. hourly) and rolling mean PM2.5 values for a specific monitor.

Usage

```
monitor_rollingMeanPlot(ws_monitor, monitorID = NULL, width = 3,
  align = "center", data.thresh = 75, tlim = NULL, ylim = NULL,
  localTime = TRUE, shadedNight = FALSE, aqiLines = TRUE,
  gridHorizontal = FALSE, grid24hr = FALSE, grid3hr = FALSE,
  showLegend = TRUE)
```

Arguments

ws_monitor	ws_monitor object
monitorID	Monitor ID for a specific monitor in the ws_monitor object (optional if only one monitor in the ws_monitor object).
width	Number of periods to average (e.g. for hourly data, width = 24 plots 24-hour rolling means).
align	Alignment of averaging window relative to point being calculated; one of "left center right".
data.thresh	Minimum number of valid observations required as a percent of width; NA is returned if insufficient valid data to calculate. mean
tlim	Optional vector with start and end times (integer or character representing YYYYMM-MDD[HH]).
ylim	y limits for the plot.
localTime	Logical specifying whether tlim is in local time or UTC.
shadedNight	Add nighttime shading.
aqiLines	Horizontal lines indicating AQI levels.
gridHorizontal	Add dashed horizontal grid lines.
grid24hr	Add dashed grid lines at day boundaries.
grid3hr	Add dashed grid lines every 3 hours.
showLegend	Include legend in top left.

Details

- align = "left": Forward roll, using hour of interest and the (width-1) subsequent hours (e.g. 3-hr left-aligned roll for Hr 5 will consist of average of Hrs 5, 6 and 7)
- align = "right": Backwards roll, using hour of interest and the (width-1) prior hours (e.g. 3-hr right-aligned roll for Hr 5 will consist of average of Hrs 3, 4 and 5)
- align = "center" for odd width: Average of hour of interest and (width-1)/2 on either side (e.g. 3-hr center-aligned roll for Hr 5 will consist of average of Hrs 4, 5 and 6)
- align = "center" for even width: Average of hour of interest and (width/2)-1 hours prior and width/2 hours after (e.g. 4-hr center-aligned roll for Hr 5 will consist of average of Hrs 4, 5, 6 and 7)

Note

This function attempts to provide a 'publication ready' rolling mean plot.

Examples

```
N_M <- Northwest_Megafires
Roseburg <- monitor_subset(N_M, tlim = c(20150821, 20150831),
                           monitorIDs = c("410190002_01"))
monitor_rollingMeanPlot(Roseburg, shadedNight = TRUE)
```

monitor_scaleData *Scale ws_monitor Data*

Description

Scale the data in a *ws_monitor* object by mutiplying it with *factor*.

Usage

```
monitor_scaleData(ws_monitor, factor)
```

Arguments

<i>ws_monitor</i>	<i>ws_monitor</i> object
<i>factor</i>	numeric used to scale the data

Value

A *ws_monitor* object with scaled data.

Examples

```
wa <- monitor_subset(Northwest_Megafires, stateCodes='WA')
wa_zero <- monitor_scaleData(wa, 3.4)
```

monitor_stamenmap *Create a static map of ws_monitor object*

Description

Plots amap showing *ws_monitor* locations and values.

#' Available maotypes include:

- terrain
- toner
- watercolor

See `staticmap_getStamenmapBrick` for details.

If `centerLon`, `centerMap` or `zoom` are not specified, appropriate values will be calculated using data from the `ws_monitor$meta` dataframe.

Usage

```
monitor_stamenmap(ws_monitor, slice = get("max"),
  breaks = AQI$breaks_24, colors = AQI$colors, width = 640,
  height = 640, centerLon = NULL, centerLat = NULL, zoom = NULL,
  maptype = "terrain", grayscale = FALSE, rasterBrick = NULL,
  cex = par("cex") * 2, pch = 16, ...)
```

Arguments

ws_monitor	ws_monitor object
slice	either a time index or a function used to collapse the time axis – defaults to get('max')
breaks	set of breaks used to assign colors
colors	a set of colors for different levels of air quality data determined by breaks
width	width of image, in pixels
height	height of image, in pixels
centerLon	map center longitude
centerLat	map center latitude
zoom	map zoom level
maptype	map type
grayscale	logical, if TRUE the colored map tile is rendered into a black & white image
rasterBrick	optional RGB rasterBrick object returned from staticmap_get~Brick)
cex	character expansion for points
pch	plotting character for points
...	arguments passed on to staticmap_plotRasterBrick() (e.g. destfile, cex, pch, etc.)

Value

Plots a map loaded from arcGIS REST with points for each monitor.

See Also

```
staticmap_getStamenmapBrick
staticmap_plotRasterBrick
```

Examples

```
## Not run:
N_M <- Northwest_Megafires
# monitor_leaflet(N_M) # to identify Spokane monitorIDs
Spokane <- monitor_subsetBy(N_M, stringr::str_detect(N_M$meta$monitorID, '^53063'))
Spokane <- monitor_subset(Spokane, tlim=c(20150815, 20150831))
monitor_stamenmap(Spokane)

## End(Not run)
```

monitor_staticmap *Create a static map of ws_monitor object*

Description

Plots a map showing *ws_monitor* locations and values.

See `staticmap_getRasterBrick` for a list of available `maptype` options.

If `centerLon`, `centerMap` or `zoom` are not specified, appropriate values will be calculated using data from the `ws_monitor$meta` dataframe.

Usage

```
monitor_staticmap(ws_monitor, slice = get("max"),
  breaks = AQI$breaks_24, colors = AQI$colors, width = 640,
  height = 640, centerLon = NULL, centerLat = NULL, zoom = NULL,
  maptype = "terrain", grayscale = FALSE, rasterBrick = NULL,
  cex = par("cex") * 2, pch = 16, ...)
```

Arguments

<code>ws_monitor</code>	<i>ws_monitor</i> object
<code>slice</code>	either a time index or a function used to collapse the time axis – defaults to <code>get('max')</code>
<code>breaks</code>	set of breaks used to assign colors
<code>colors</code>	a set of colors for different levels of air quality data determined by <code>breaks</code>
<code>width</code>	width of image, in pixels
<code>height</code>	height of image, in pixels
<code>centerLon</code>	map center longitude
<code>centerLat</code>	map center latitude
<code>zoom</code>	map zoom level
<code>maptype</code>	map type
<code>grayscale</code>	logical, if TRUE the colored map tile is rendered into a black & white image
<code>rasterBrick</code>	optional RGB <code>rasterBrick</code> object returned from <code>staticmap_get~Brick</code>
<code>cex</code>	character expansion for points
<code>pch</code>	plotting character for points
<code>...</code>	arguments passed on to <code>staticmap_plotRasterBrick()</code> (e.g. <code>destfile</code> , <code>cex</code> , <code>pch</code> , etc.)

Value

A plot with a basemap and colored dots for each monitor.

See Also

```
staticmap_getStamenmapBrick
staticmap_plotRasterBrick
```

Examples

```
## Not run:
N_M <- Northwest_Megafires
# monitor_leaflet(N_M) # to identify Spokane monitorIDs
Spokane <- monitor_subsetBy(N_M, stringr::str_detect(N_M$meta$monitorID, '^53063'))
Spokane <- monitor_subset(Spokane, tlim=c(20150815, 20150831))
monitor_staticmap(Spokane)

## End(Not run)
```

monitor_subset	<i>Subset ws_monitor Object</i>
----------------	---------------------------------

Description

Creates a subset *ws_monitor* based on one or more optional input parameters. If any input parameter is not specified, that parameter will not be used to subset *ws_monitor*.

Usage

```
monitor_subset(ws_monitor, xlim = NULL, ylim = NULL, tlim = NULL,
              vlim = NULL, monitorIDs = NULL, stateCodes = NULL,
              countryCodes = NULL, dropMonitors = TRUE, timezone = "UTC")
```

Arguments

ws_monitor	<i>ws_monitor</i> object
xlim	optional vector with low and high longitude limits
ylim	optional vector with low and high latitude limits
tlim	optional vector with start and end times (integer or character representing YYYYMM-DD[HH] or POSIXct)
vlim	optional vector with low and high data value limits
monitorIDs	optional vector of monitor IDs used to filter the data
stateCodes	optional vector of state codes used to filter the data
countryCodes	optional vector of country codes used to filter the data
dropMonitors	flag specifying whether to remove monitors with no data
timezone	Olson timezone passed to <code>link{parseDatetime}</code> when parsing numeric tlim

Details

By default, this function will return a `ws_monitor` object whose `data` dataframe has the same number of columns as the incoming dataframe, unless any of the columns consist of all NAs, in which case such columns will be removed (*e.g.* if there are no valid data for a specific monitor after subsetting by `tlim` or `vlim`). If `dropMonitors=FALSE`, columns that consist of all NAs will be retained.

Value

A `ws_monitor` object with a subset of `ws_monitor`.

Examples

```
N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150701,20150731))
xlim <- c(-124.73, -122.80)
ylim <- c(47.20, 48.40)
Olympic_Peninsula <- monitor_subset(N_M, xlim, ylim)
monitor_map(Olympic_Peninsula, cex=2)
rect(xlim[1], ylim[1], xlim[2], ylim[2], col=adjustcolor('black',0.1))
```

monitor_subsetBy *Subset ws_monitor Object with a Filter*

Description

The incoming `ws_monitor` object is filtered according to `filter`. Either meta data or actual data can be filtered.

Usage

```
monitor_subsetBy(ws_monitor, filter)
```

Arguments

`ws_monitor` *ws_monitor* object
`filter` a filter to use on the `ws_monitor` object

Value

A `ws_monitor` object with a subset of the input `ws_monitor` object.

Examples

```
N_M <- Northwest_Megafires
boise_tz <- monitor_subsetBy(N_M, timezone == 'America/Boise')
boise_tz_very_unhealthy <- monitor_subsetBy(boise_tz, data > AQI$breaks_24[5])
boise_tz_very_unhealthy$meta$siteName
```

 monitor_subsetByDistance

Subset ws_monitor Object by Distance from Target Location

Description

Subsets `ws_monitor` to include only those monitors (or grid cells) within a certain radius of a target location. If no monitors (or grid cells) fall within the specified `radius`, `ws_monitor$data` and `ws_monitor$meta` are set to `NULL`.

When `count` is used, a `ws_monitor` object is created containing **up to** `count` monitors, ordered by increasing distance from the target location. Thus, note that the number of monitors (or grid cells) returned may be less than the specified `count` value if fewer than `count` monitors (or grid cells) are found within the specified `radius` of the target location.

Usage

```
monitor_subsetByDistance(ws_monitor, longitude = NULL, latitude = NULL,
  radius = 50, count = NULL)
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object
<code>longitude</code>	target longitude from which the radius will be calculated
<code>latitude</code>	target latitude from which the radius will be calculated
<code>radius</code>	distance (km) of radius from target location – default=300
<code>count</code>	number of grid cells to return

Value

A `ws_monitor` object with monitors near a location.

See Also

`monitorDistance`

Examples

```
## Not run:
# Napa Fires -- October, 2017
ca <- airnow_loadAnnual(2017) %>%
  monitor_subset(tlim=c(20171001,20171101), stateCodes='CA')
Vallejo <- monitor_subset(ca, monitorIDs='060950004_01')
Napa_Fires <- monitor_subsetByDistance(ca,
  longitude = Vallejo$meta$longitude,
  latitude = Vallejo$meta$latitude,
  radius = 50)

monitor_leaflet(Napa_Fires)
```

```
## End(Not run)
```

```
monitor_subsetData Subset ws_monitor Object 'data' Dataframe
```

Description

Subsets a *ws_monitor* object's data dataframe by removing any monitors that lie outside the specified ranges of time and values and that are not mentioned in the list of monitorIDs.

If *tlim* or *vlim* is not specified, it will not be used in the subsetting.

Intended for use by the *monitor_subset* function.

Usage

```
monitor_subsetData(data, tlim = NULL, vlim = NULL, monitorIDs = NULL,
  dropMonitors = FALSE, timezone = "UTC")
```

Arguments

<i>data</i>	<i>ws_monitor</i> object data dataframe
<i>tlim</i>	optional vector with start and end times (integer or character representing YYYYMMDD[HH] or POSIXct)
<i>vlim</i>	optional vector with low and high data value limits
<i>monitorIDs</i>	optional vector of monitorIDs
<i>dropMonitors</i>	flag specifying whether to remove columns – defaults to FALSE
<i>timezone</i>	Olson timezone passed to <code>link{parseDatetime}</code> when parsing numeric <i>tlim</i>

Details

By default, filtering by *tlim* or *vlim* will always return a dataframe with the same number of columns as the incoming dataframe. If *dropMonitors*=TRUE, columns will be removed if there are not valid data for a specific monitor after subsetting.

Filtering by *vlim* is open on the left and closed on the right, i.e.

```
x > vlim[1] & x <= vlim[2]
```

Value

A *ws_monitor* object data dataframe, or NULL if filtering removes all monitors.

monitor_subsetMeta *Subset ws_monitor Object 'meta' Dataframe*

Description

Subsets the `ws_monitor$data` dataframe by removing any monitors that lie outside the geographical ranges specified (i.e. outside of the given longitudes and latitudes and/or states) and that are not mentioned in the list of `monitorIDs`.

If any parameter is not specified, that parameter will not be used in the subsetting.

Intended for use by the `monitor_subset` function.

Usage

```
monitor_subsetMeta(meta, xlim = NULL, ylim = NULL, stateCodes = NULL,
  countryCodes = NULL, monitorIDs = NULL)
```

Arguments

<code>meta</code>	<code>ws_monitor</code> object <code>meta</code> dataframe
<code>xlim</code>	optional vector with low and high longitude limits
<code>ylim</code>	optional vector with low and high latitude limits
<code>stateCodes</code>	optional vector of <code>stateCodes</code>
<code>countryCodes</code>	optional vector of <code>countryCodes</code>
<code>monitorIDs</code>	optional vector of <code>monitorIDs</code>

Details

Longitudes must be specified in the domain [-180,180].

Value

A `ws_monitor` object `meta` dataframe, or `NULL` if filtering removes all monitors.

`monitor_timeAverage`

Calculate Time Averages

Description

This function extracts the `data` dataframe from `ws_monitor` object and renames the `'datetime'` column so that it can be processed by the **openair** package's `timeAverage()` function. (See that function for details.)

Usage

```
monitor_timeAverage(ws_monitor, ...)
```

Arguments

```
ws_monitor    ws_monitor object
...           additional arguments to be passed to openair::timeAverage()
```

Value

A *ws_monitor* object with data that have been processed by `openair::timeAverage()`.

Examples

```
C_V <- monitor_subset(Carmel_Valley, tlim=c(2016080800,2016081023),
                     timezone='America/Los_Angeles')
C_V_3hourly <- monitor_timeAverage(C_V, avg.time="3 hour")
head(C_V$data, n=15)
head(C_V_3hourly$data, n=5)
```

monitor_timeInfo *Get time related information for a monitor*

Description

Calculate the local time for the monitor, as well as sunrise, sunset and solar noon times, and create several temporal masks.

The returned dataframe will have as many rows as the length of the incoming UTC `time` vector and will contain the following columns:

- `localStdTime_UTC` – UTC representation of local **standard** time
- `daylightSavings` – logical mask = TRUE if daylight savings is in effect
- `localTime` – local clock time
- `sunrise` – time of sunrise on each `localTime` day
- `sunset` – time of sunset on each `localTime` day
- `solarnoon` – time of solar noon on each `localTime` day
- `day` – logical mask = TRUE between sunrise and sunset
- `morning` – logical mask = TRUE between sunrise and `solarnoon`
- `afternoon` – logical mask = TRUE between `solarnoon` and sunset
- `night` – logical mask = opposite of day

Usage

```
monitor_timeInfo(ws_monitor = NULL, monitorID = NULL)
```

Arguments

`ws_monitor` *ws_monitor* object.

`monitorID` Monitor ID for a specific monitor in `ws_monitor` – optional if `ws_monitor` only has one monitor.

Details

While the **lubridate** package makes it easy to work in local timezones, there is no easy way in R to work in "Local Standard Time" (LST) as is often required when working with air quality data. EPA regulations mandate that daily averages be calculated based on LST.

The `localStdTime_UTC` is primarily for use internally and provides an important tool for creating LST daily averages and LST axis labeling.

Value

A dataframe with times and masks.

Examples

```
carmel <- monitor_subset(Carmel_Valley, tlim = c(20160801,20160810))

# Create timeInfo object for this monitor
ti <- monitor_timeInfo(carmel)

# Subset the data based on day/night masks
data_day <- carmel$data[ti$day,]
data_night <- carmel$data[ti$night,]

# Build two monitor objects
carmel_day <- list(meta = carmel$meta, data = data_day)
carmel_night <- list(meta = carmel$meta, data = data_night)

# Plot them
monitor_timeseriesPlot(carmel_day, shadedNight = TRUE, pch = 8, col = 'goldenrod')
monitor_timeseriesPlot(carmel_night, pch = 16, col = 'darkblue', add = TRUE)
```

```
monitor_timeseriesPlot
      Create Timeseries Plot
```

Description

Creates a time series plot of PM2.5 data from a *ws_monitor* object (see note below). Optional arguments color code by AQI index, add shading to indicate nighttime, and adjust the time display (local vs. UTC).

When a named `style` is used, some graphical parameters will be overridden. Available styles include:

- `aqidots`— hourly values are individually colored by 24-hr AQI levels
- `gnats`— semi-transparent dots like a cloud of gnats

Usage

```
monitor_timeseriesPlot(ws_monitor, monitorID = NULL, tlim = NULL,
  localTime = TRUE, style = NULL, shadedNight = FALSE, add = FALSE,
  gridPos = "", gridCol = "black", gridLwd = 1, gridLty = "solid",
  dayLwd = 0, hourLwd = 0, hourInterval = 6, ...)
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object.
<code>monitorID</code>	Monitor ID for one or more monitor in the <code>ws_monitor</code> object.
<code>tlim</code>	Optional vector with start and end times (integer or character representing YYYYMM-MDD[HH]).
<code>localTime</code>	Logical specifying whether <code>tlim</code> is in local time or UTC.
<code>style</code>	Custom styling, one of "aqidots".
<code>shadedNight</code>	Add nighttime shading.
<code>add</code>	Logical specifying whether to add to the current plot.
<code>gridPos</code>	Position of grid lines either "over", "under" (" " for no grid lines).
<code>gridCol</code>	Grid line color.
<code>gridLwd</code>	Grid line width.
<code>gridLty</code>	Grid line type.
<code>dayLwd</code>	Day marker line width.
<code>hourLwd</code>	Hour marker line width.
<code>hourInterval</code>	Interval for grid (max = 12).
<code>...</code>	Additional arguments to be passed to <code>points()</code> .

Note

Remember that a `ws_monitor` object can contain data from more than one monitor, and thus, this function may produce a time series of data from multiple monitors. To plot a time series of an individual monitor's data, specify a single `monitorID`.

Examples

```
N_M <- Northwest_Megafires
# monitor_leaflet(N_M) # to identify Spokane monitorIDs
Spokane <- monitor_subsetBy(
  N_M,
  stringr::str_detect(N_M$meta$monitorID, "^53063")
)

monitor_timeseriesPlot(Spokane, style = "gnats")
```

```

title("Spokane PM2.5 values, 2015")
monitor_timeseriesPlot(
  Spokane,
  tlim = c(20150801, 20150831),
  style = "aqidots",
  pch = 16
)
addAQILegend()
title("Spokane PM2.5 values, August 2015")
monitor_timeseriesPlot(
  Spokane,
  tlim = c(20150821, 20150828),
  shadedNight = TRUE,
  style = "gnats"
)
abline(h = AQI$breaks_24, col = AQI$colors, lwd = 2)
addAQILegend()
title("Spokane PM2.5 values, August 2015")

```

monitor_toTidy *Convert 'ws_monitor' data to a tidy format*

Description

Changes write-optimized 'ws_monitor' formatted data into a read-optimized 'tidy' format that is useful for 'tidyverse' functions. If the given data is already in a tidy format, it is returned as is.

Usage

```
monitor_toTidy(data = NULL)
```

Arguments

data Data to potentially convert.

Value

'Tidy' formatted 'ws_monitor' data.

Examples

```

ws_monitor <- monitor_subset(
  Northwest_Megafires,
  monitorIDs = c('530470009_01', '530470010_01')
)

ws_monTidy <- monitor_toTidy(ws_monitor)

## Not run:
ws_monTidy2 <- monitor_toTidy(ws_monTidy)

```

```
## End(Not run)
```

monitor_trim	<i>Trim ws_monitor Time Axis to Remove NA Periods From Beginning and End</i>
--------------	--

Description

Trims the time axis of a *ws_monitor* object to exclude timestamps prior to the first and after the last valid datapoint for any monitor.

Usage

```
monitor_trim(ws_monitor)
```

Arguments

ws_monitor *ws_monitor* object

Value

A *ws_monitor* object with missing data trimmed.

Examples

```
## Not run:
sm13 <- wrcc_createMonitorObject(20150101, 20151231, unitID='sm13')
sm13$meta[,c('stateCode', 'countyName', 'siteName', 'monitorID')]
Deschutes <- monitor_subset(sm13, monitorIDs='lon_.121.453_lat_43.878_wrcc.sm13')
Deschutes <- monitor_trim(Deschutes)
monitor_dailyBarplot(Deschutes)

## End(Not run)
```

monitor_writeCSV	<i>Write monitor data as CSV</i>
------------------	----------------------------------

Description

Prints out the contents of the *ws_monitor* object as CSV. By default, the output is a text string with "human readable" CSV that includes both *meta* and *data*. When saved as a file, this format is useful for point-and-click spreadsheet users who want to have everything on a single sheet.

To obtain machine parseable CSV strings you can use *metaOnly* or *dataOnly* which are mutually exclusive but which return CSV strings that can be automatically ingested.

By default, the CSV formatted text is returned invisibly but not saved to a file unless *saveFile* is specified.

Usage

```
monitor_writeCSV(ws_monitor, saveFile = NULL, metaOnly = FALSE,  
  dataOnly = FALSE, quietly = TRUE)
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object
<code>saveFile</code>	optional filename where CSV will be written
<code>metaOnly</code>	flag specifying whether to return <code>ws_monitor\$meta</code> only as a machine parseable CSV
<code>dataOnly</code>	flag specifying whether to return <code>ws_monitor\$data</code> only as a machine parseable CSV
<code>quietly</code>	do not print to console, just return the string representation of the CSV

Note

This function wraps the `monitor_print` function but defaults to `quietly = FALSE`.

Examples

```
data("Carmel_Valley")  
Carmel_Valley <- monitor_subset(Carmel_Valley, tlim = c(20160802,20160803))  
monitor_print(Carmel_Valley)  
monitor_print(Carmel_Valley, metaOnly = TRUE)  
monitor_print(Carmel_Valley, dataOnly = TRUE)
```

```
monitor_writeCurrentStatusGeoJSON  
  Write current monitor data to geojson file
```

Description

Writes a geoJSON file containing current monitor data. For details on what is included, see `monitor_getCurrentStatus`.

Usage

```
monitor_writeCurrentStatusGeoJSON(ws_monitor, filename,  
  datetime = lubridate::now("UTC"), properties = NULL,  
  propertyNames = NULL, metadataList = list())
```

Arguments

<code>ws_monitor</code>	<code>ws_monitor</code> object.
<code>filename</code>	Filename where geojson file will be saved.
<code>datetime</code>	Time to which data will be 'current' (integer or character representing YYYYMM-MDDHH or <code>POSIXct</code> . If not <code>POSIXct</code> , interpreted as UTC time). So if <code>datetime</code> is 3 hours ago, a dataframe with the most current data from 3 hours ago will be returned.
<code>properties</code>	Optional character vector of properties to include for each monitor in geoJSON. If NULL all are included. May include any <code>ws_monitor</code> metadata and additional columns generated in <code>monitor_getCurrentStatus</code> .
<code>propertyNamees</code>	Optional character vector supplying custom names for properties in geoJSON. If NULL or different length than <code>properties</code> defaults will be used.
<code>metadataList</code>	List of top-level foreign members to include. May include nested lists as long as they can be converted into JSON using <code>jsonlite::toJSON()</code> . For more information on what can be included see https://tools.ietf.org/html/rfc7946#section-6.1 ²⁹ .

Value

Invisibly returns geoJSON string.

Examples

```
## Not run:
wa <-
  monitor_loadLatest() %>%
  monitor_subset(stateCodes = "WA")
wa_current_geojson <- monitor_writeCurrentStatusGeoJSON(wa, "wa_monitors.geojson")
wa_current_list <- jsonlite::fromJSON(wa_current_geojson)
wa_spdf <- rgdal::readOGR(dsn = "wa_monitors.geojson", layer = "OGRGeoJSON")
points(wa_spdf)

## End(Not run)
```

Northwest_Megafires

Northwest Megafires Example Dataset

Description

In the summer of 2015 Washington state had several catastrophic wildfires that led to many days of heavy smoke in eastern Washington, Oregon and northern Idaho. The `Northwest_Megafires` dataset contains AirNow ambient monitoring data for the Pacific Northwest from May 31 through November 01, 2015 (UTC). Data are stored as a `ws_monitor` object and are used in many examples in the package documentation.

²⁹<https://tools.ietf.org/html/rfc7946#section-6.1>

Format

A list with two elements

Details

Northwest_Megafires example dataset

parseDatetime *Parse Datetime Strings*

Description

Transforms numeric and string representations of Ymd[HMS] datetimes to POSIXct format.

Ymd, YmdH, YmdHM, and YmdHMS formats are understood, where:

* Y - four digit year * m - decimal number (1-12, 01-12) or english name month (October, oct.) *
 d - decimal number day of the month (0-31 or 01-31) * H - decimal number hours (0-24 or 00-24)
 * M - decimal number minutes (0-59 or 00-59) * S - decimal number seconds (0-61 or 00-61)

This allows for mixed inputs. For example, 20181012130900, "2018-10-12-13-09-00", and "2018 Oct. 12 13:09:00" will all be converted to the same POSIXct datetime. The incoming datetime vector does not need to have a homogeneous format either – "20181012" and "2018-10-12 13:09" can exist in the same vector without issue. All incoming datetimes will be interpreted in the specified timezone.

If datetime is a POSIXct it will be returned unmodified, and formats not recognized will be returned as NA.

Usage

```
parseDatetime(datetime, timezone = "UTC", expectAll = FALSE)
```

Arguments

datetime	vector of character or integer datetimes in Ymd[HMS] format (or POSIXct).
timezone	Olson timezone at the location of interest (default "UTC").
expectAll	Logical value determining if the function should fail if any elements fail to parse (default FALSE).

Value

POSIXct datetimes.

PWFSLSmoke Conventions

Within the PWFSLSmoke package, datetimes not in POSIXct format are represented as decimal values with no separation (ex: 20181012, 20181012130900), either as numerics or strings.

Implementation

parseDatetime is essentially a wrapper around parse_date_time, handling which formats we want to account for.

See Also

parse_date_time for implementation details.

Examples

```
starttime <- parseDatetime(2015080718, timezone = "America/Los_Angeles")
datetimes <- parseDatetime(c("20181014 12", "20181015 12", "20181016 12"))

## Not run:
badInput <- c("20181013", NA, "20181015", "181016", "10172018")

# This will return a vector with the date that were able to parse
parseDatetime(badInput, expectAll = FALSE)

# This will return an error, since some non-NA indices didn't parse
parseDatetime(badInput, expectAll = TRUE)

## End(Not run)
```

```
rawPlot_pollutionRose
```

Create Pollution Rose Plot from a Raw Dataframe

Description

Create pollution rose plot from an enhanced raw dataframe. This function is based on `openair::pollutionRose()`. If normalized, black line indicates frequency by direction.

Usage

```
rawPlot_pollutionRose(df, parameter = "pm25", tlim = NULL,
  localTime = TRUE, normalize = FALSE, ...)
```

Arguments

df	enhanced, raw dataframe as created by the <code>raw_enhance()</code> function
parameter	parameter to plot
tlim	optional vector with start and end times (integer or character representing YYYYM-MDD[HH])
localTime	logical specifying whether tlim is in local time or UTC
normalize	normalize slices to fill entire area, allowing for easier comparison of counts of magnitudes by direction
...	additional arguments to pass on to <code>openair::pollutionRose()</code>

Note

If more than one timezone is found, `localTime` is ignored and UTC is used.

Examples

```
## Not run:
raw <- airts_createRawDataframe(20160901, 20161015, 'USFS', 1012)
raw <- raw_enhance(raw)
rawPlot_pollutionRose(raw)

## End(Not run)
```

```
rawPlot_timeOfDaySpaghetti
```

Create Time of Day Spaghetti Plot from a Raw Dataframe

Description

Spaghetti Plot that shows data by hour-of-day.

Usage

```
rawPlot_timeOfDaySpaghetti(df, parameter = "pm25", tlim = NULL,
  shadedNight = TRUE, meanCol = "black", meanLwd = 4, meanLty = 1,
  highlightDates = c(), highlightCol = "dodgerblue", ...)
```

Arguments

<code>df</code>	enhanced, raw dataframe as created by the <code>raw_enhance()</code> function
<code>parameter</code>	variable to be plotted
<code>tlim</code>	optional vector with start and end times (integer or character representing YYYYMMDD[HH])
<code>shadedNight</code>	add nighttime shading
<code>meanCol</code>	color used for the mean line (use NA to omit the mean)
<code>meanLwd</code>	line width used for the mean line
<code>meanLty</code>	line type used for the mean line
<code>highlightDates</code>	dates to be highlighted in YYYYMMDD format
<code>highlightCol</code>	color used for highlighted days
<code>...</code>	additional graphical parameters are passed to the <code>lines()</code> function for day lines

Examples

```
## Not run:
raw <- airts_createRawDataframe(20160901, 20161015, 'USFS', 1012)
raw <- raw_enhance(raw)
rawPlot_timeOfDaySpaghetti(raw, parameter="temperature")

## End(Not run)
```

rawPlot_timeseries *Create Timeseries Plot from a Raw Dataframe*

Description

Creates a plot of raw monitoring data as generated using `raw_enhance()`.

Other options for `parameter` include "temperature", "humidity", "windSpeed", "windDir", "pressure" or any of the other raw parameters (try `names(df)` to see list of options)

Usage

```
rawPlot_timeseries(df, parameter = "pm25", tlim = NULL,
  localTime = TRUE, shadedNight = TRUE, shadedBackground = NULL,
  sbLwd = 1, add = FALSE, gridPos = "", gridCol = "black",
  gridLwd = 1, gridLty = "solid", dayLwd = 0, hourLwd = 0,
  hourInterval = 6, ...)
```

Arguments

<code>df</code>	enhanced, raw dataframe as created by the <code>raw_enhance()</code> function
<code>parameter</code>	raw parameter to plot
<code>tlim</code>	optional vector with start and end times (integer or character representing YYYYM-MDD[HH])
<code>localTime</code>	logical specifying whether <code>tlim</code> is in local time or UTC
<code>shadedNight</code>	add nighttime shading
<code>shadedBackground</code>	add vertical lines for a second parameter
<code>sbLwd</code>	shaded background line width
<code>add</code>	logical specifying whether to add to the current plot
<code>gridPos</code>	position of grid lines either 'over', 'under' or '' for no grid lines
<code>gridCol</code>	grid line color
<code>gridLwd</code>	grid line width
<code>gridLty</code>	grid line type
<code>dayLwd</code>	day marker line width
<code>hourLwd</code>	hour marker line width
<code>hourInterval</code>	interval for grid (max=12)
<code>...</code>	additional arguments to pass to <code>lines()</code> function

Details

Note that for multiple deployments, `shadedNight` defaults to use the lat/lon for the first deployment, which in theory could be somewhat unrepresentative, such as if deployments have a large range in latitude.

Note

If more than one timezone is found, `localTime` is ignored and UTC is used.

`rawPlot_windRose` *Create Wind Rose Plot from a Raw Dataframe*

Description

Create wind rose plot from `raw_enhance` object. Based on `openair::windRose()`.

Usage

```
rawPlot_windRose(df, tlim = NULL, localTime = TRUE, ...)
```

Arguments

<code>df</code>	enhanced, raw dataframe as created by the <code>raw_enhance()</code> function
<code>tlim</code>	optional vector with start and end times (integer or character representing YYYYMM-MDD[HH])
<code>localTime</code>	logical specifying whether <code>tlim</code> is in local time or UTC
<code>...</code>	additional arguments to pass on to <code>openair::windRose()</code>

Note

If more than one timezone is found, `localTime` is ignored and UTC is used.

Examples

```
## Not run:
raw <- aairsis_createRawDataframe(20160901, 20161015, provider='USFS', unitID=1012)
raw <- raw_enhance(raw)
rawPlot_windRose(raw)

## End(Not run)
```

`raw_enhance`*Process Raw Monitoring Data to Create raw_enhance Object*

Description

Processes raw monitor data to add a uniform time axis and consistent data columns that can be handled by various `raw~` functions. All original raw data is retained, and the following additional columns are added:

- `dataSource`
- `longitude`
- `latitude`
- `temperature`
- `humidity`
- `windSpeed`
- `windDir`
- `pressure`
- `pm25`

The `datetime` column in the incoming dataframe may have missing hours. This time axis is expanded to a uniform, hourly axes with missing data fields added for data columns.

Usage

```
raw_enhance(df)
```

Arguments

`df` raw monitor data, as created by `airsis_createRawDataframe` or `wrcc_createRawDataframe`

Value

Dataframe with original raw data, plus new columns with raw naming scheme for downstream use.

Examples

```
## Not run:
library(MazamaWebUtils)
df <- airsis_createRawDataframe(startdate=20160901, enddate=20161015, provider='USFS', unitI
df <- raw_enhance(df)
rawPlot_timeseries(df, tlim=c(20160908,20160917))

## End(Not run)
```

`raw_getHighlightDates`*Return Day Stamps for Values Above a Threshold*

Description

Returna list of dates in YYYYMMDD format where the dataVar is within highlightRange.

Usage

```
raw_getHighlightDates(df, dataVar, tzone = NULL,  
  highlightRange = c(1e+12, Inf))
```

Arguments

df	dataframe with datetime column in UTC
dataVar	variable to be evaluated
tzone	timezone where data were collected
highlightRange	range of values of to be highlighted

Examples

```
## Not run:  
raw <- airsis_createRawDataframe(startdate = 20160901, provider = 'USFS',unitID = '1033')  
raw <- raw_enhance(raw)  
highlightRange <- c(50,Inf)  
dataVar <- 'pm25'  
tzone <- "America/Los_Angeles"  
highlightDates <- raw_getHighlightDates(raw,dataVar,tzone,highlightRange)  
rawPlot_timeOfDaySpaghetti(df=raw,highlightDates = highlightDates)  
  
## End(Not run)
```

`setEsriToken`*Set ESRI Token*

Description

Sets the current esriToken.

Usage

```
setEsriToken(token)
```

Arguments

token ESRI token used when interacting with ESRI location services

Value

Silently returns previous value of esriToken.

See Also

addEsriAddress

getEsriToken

esriToken

setGoogleApiKey *Set Google API Key*

Description

Sets the current Google API key.

Usage

```
setGoogleApiKey (key)
```

Arguments

key Google API key used when interacting with Google location services

Value

Silently returns previous value of googleApiKey.

See Also

addGoogleAddress

addGoogleElevation

getGoogleApiKey

googleApiKey

skill_confusionMatrix

Confusion Matrix Statistics

Description

Measurements of categorical forecast accuracy have a long history in weather forecasting. The standard approach involves making binary classifications (detected/not-detected) of predicted and observed data and combining them in a binary contingency table known as a *confusion matrix*.

This function creates a `confusion matrix` from predicted and observed values and calculates a wide range of common statistics including:

- TP (true positive)
- FP (false positive) (type I error)
- FN (false negative) (type II error)
- TN (true negative)
- TPRate (true positive rate) = sensitivity = recall = $TP / (TP + FN)$
- FPRate (false positive rate) = $FP / (FP + TN)$
- FNRate (false negative rate) = $FN / (TP + FN)$
- TNRate (true negative rate) = specificity = $TN / (FP + TN)$
- accuracy = proportionCorrect = $(TP + TN) / \text{total}$
- errorRate = $1 - \text{accuracy} = (FP + FN) / \text{total}$
- falseAlarmRatio = PPV (positive predictive value) = precision = $TP / (TP + FP)$
- FDR (false discovery rate) = $FP / (TP + FP)$
- NPV (negative predictive value) = $TN / (TN + FN)$
- FOR (false omission rate) = $FN / (TN + FN)$
- f1_score = $(2 * TP) / (2 * TP + FP + FN)$
- detectionRate = TP / total
- baseRate = detectionPrevalence = $(TP + FN) / \text{total}$
- probForecastOccurance = prevalence = $(TP + FP) / \text{total}$
- balancedAccuracy = $(TPRate + TNRate) / 2$
- expectedAccuracy = $((TP + FP) * (TP + FN) / \text{total}) + ((FP + TN) * \text{sum}(FN + TN) / \text{total}) / \text{total}$
- heidkeSkill = kappa = $(\text{accuracy} - \text{expectedAccuracy}) / (1 - \text{expectedAccuracy})$
- bias = $(TP + FP) / (TP + FN)$
- hitRate = $TP / (TP + FN)$
- falseAlarmRate = $FP / (FP + TN)$
- pierceSkill = $((TP * TN) - (FP * FN)) / ((FP + TN) * (TP + FN))$
- criticalSuccess = $TP / (TP + FP + FN)$
- oddsRatioSkill = yulesQ = $((TP * TN) - (FP * FN)) / ((TP * TN) + (FP * FN))$

Usage

```
skill_confusionMatrix(predicted, observed, FPCost = 1, FNCost = 1,
  lightweight = FALSE)
```

Arguments

predicted	logical vector of predicted values
observed	logical vector of observed values
FPCost	cost associated with false positives (type I error)
FNCost	cost associated with false negatives (type II error)
lightweight	flag specifying creation of a return list without derived metrics

Value

List containing a table of `confusion matrix` values and a suite of derived metrics.

References

Simple Guide to Confusion Matrix Terminology³⁰

See Also

skill_ROC
skill_ROCPlot

Examples

```
predicted <- sample(c(TRUE,FALSE), 1000, replace=TRUE, prob=c(0.3,0.7))
observed <- sample(c(TRUE,FALSE), 1000, replace=TRUE, prob=c(0.3,0.7))
cm <- skill_confusionMatrix(predicted, observed)
print(cm)
```

skill_ROC

ROC Curve

Description

This function calculates an ROC dataframe of TPR, FPR, and Cost for a range of thresholds as well as the area under the ROC curve.

Usage

```
skill_ROC(predicted, observed, t1Range = NULL, t2 = NULL, n = 101)
```

³⁰<http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/>

Arguments

predicted	vector of predicted values (or a <i>ws_monitor</i> object with a single location)
observed	vector of observed values (or a <i>ws_monitor</i> object with a single location)
t1Range	lo and high values used to generate test thresholds for classifying predicted data
t2	used to classify observed data
n	number of test thresholds in ROC curve

Value

List containing an `roc` matrix and the `auc` area under the ROC curve.

References

Receiver Operating Characteristic³¹

See Also

`skill_confusionMatrix`

`skill_ROCPlot`

Examples

```
## Not run:
# Napa Fires -- October, 2017
ca <- airnow_loadAnnual(2017) %>%
  monitor_subset(tlim = c(20171001,20171101), stateCodes = 'CA')
Vallejo <- monitor_subset(ca, monitorIDs = '060950004_01')
Napa <- monitor_subset(ca, monitorIDs = '060550003_01')
t2 <- AQI$breaks_24[4] # 'Unhealthy'
rocList <- skill_ROC(Vallejo, Napa, t1Range = c(0,100), t2 = t2)
roc <- rocList$roc
auc <- rocList$auc
plot(roc$TPR ~ roc$FPR, type = 'S')
title(paste0('Area Under Curve = ', format(auc,digits = 3)))

## End(Not run)
```

³¹https://en.wikipedia.org/wiki/Receiver_operating_characteristic

skill_ROCPlot *ROC Plot*

Description

This function plots ROC curves for a variety of observed classification thresholds.

Usage

```
skill_ROCPlot(predicted, observed, t1Range = c(0, 100), t2s = seq(10,
  100, 10), n = 101, colors = grDevices::rainbow(length(t2s)))
```

Arguments

predicted	vector of predicted values (or a <i>ws_monitor</i> object with a single location)
observed	vector of observed values (or a <i>ws_monitor</i> object with a single location)
t1Range	lo and high values used to generate test thresholds for classifying predicted data
t2s	vector of thresholds used to classify observed data
n	number of test thresholds in ROC curve
colors	vector of colors used when plotting curves

References

Receiver Operating Characteristic³²

See Also

skill_confusionMatrix
skill_ROC

Examples

```
## Not run:
# Napa Fires -- October, 2017
ca <- airnow_loadAnnual(2017) %>%
  monitor_subset(tlim = c(20171001,20171101), stateCodes = 'CA')
Vallejo <- monitor_subset(ca, monitorIDs = '060950004_01')
Napa <- monitor_subset(ca, monitorIDs = '060550003_01')
skill_ROCPlot(Vallejo, Napa)

## End(Not run)
```

³²https://en.wikipedia.org/wiki/Receiver_operating_characteristic

```
staticmap_getEsrimapBrick
```

Create a rasterBrick from an Esri tiled image server

Description

Uses the input coordinates to fetch and composite a raster from the tile server. Returns a `raster::rasterBrick` object. This can then be passed as the `rasterBrick` object to the `staticmap_plotRasterBrick()` function for plotting.

As of July 2019, this list is a handy reference to the freely available tile servers which can be previewed at the following URL:

<https://leaflet-extras.github.io/leaflet-providers/preview/>

Usage

```
staticmap_getEsrimapBrick(centerLon = NULL, centerLat = NULL,
  maptype = "world_topo", zoom = 12, width = 640, height = 640,
  bbox = NULL, maxTiles = 20, crs = sp::CRS("+init=epsg:4326"),
  tileCacheDir = tempdir())
```

Arguments

<code>centerLon</code>	Map center longitude.
<code>centerLat</code>	Map center latitude.
<code>maptype</code>	Selects the appropriate Esri tile server. Options include: <ul style="list-style-type: none"> • "world_topo" • "world_imagery" • "world_terrain" • "de_Lorme" • "world_grey" • "world_streets"
<code>zoom</code>	map Zoom level.
<code>width</code>	Width of image, in pixels.
<code>height</code>	Height of image, in pixels.
<code>bbox</code>	Bounding box vector (lonLo, latLo, lonHi, latHi). If not null, <code>centerLon</code> , <code>centerLat</code> , and <code>zoom</code> are ignored.
<code>maxTiles</code>	Maximum number of tiles to be returned. The greater the number, the slower the performance – arbitrarily set to 20 by default.
<code>crs</code>	Object of class CRS. The Coordinate Reference System (CRS) for the returned map. If the CRS of the downloaded map does not match, it will be projected to the specified CRS using <code>raster::projectRaster</code> .
<code>tileCacheDir</code>	Optional location for cached tiles.

Value

A rasterBrick object which can be plotted with `staticmap_plotRasterBrick()` or `raster::plotRGB()` and serve as a base plot.

Note

The spatial reference of the image when it is downloaded is 3857. If the `crs` argument is different, projecting may cause the size and extent of the image to differ very slightly from the input, on a scale of 1-2 pixels or 10^{-3} degrees.

If `bbox` is specified and the `bbox` aspect ratio does not match the width/height aspect ratio the extent is resized to prevent the map image from appearing stretched, so the map extent may not match the `bbox` argument exactly.

If both `zoom` and `maxTiles` are specified, `maxTiles` takes precedence. To get a specified zoom level, set `maxTiles = NULL`.

See Also

`staticmap_getRasterBrick`
`staticmap_plotRasterBrick`

Examples

```
## Not run:
rasterBrick <- staticmap_getEsrimapBrick(-122.3318, 47.668)
staticmap_plotRasterBrick(rasterBrick)

## End(Not run)
```

```
staticmap_getRasterBrick
```

Create a rasterBrick from a tiled image server

Description

Uses the input coordinates to select an appropriate method to build a `raster::rasterBrick` object. It will either use the `staticmap_getStamenmapBrick()` function or the `staticmap_getEsrimapBrick()` function. This can then be passed as the `rasterBrick` object to the `staticmap_plotRasterBrick()` function for plotting.

Usage

```
staticmap_getRasterBrick(centerLon = NULL, centerLat = NULL,
  maptype = "world_topo", zoom = 12, width = 640, height = 640,
  bbox = NULL, maxTiles = 40, crs = sp::CRS("+init=epsg:4326"),
  tileCacheDir = tempdir())
```


Arguments

centerLon	Map center longitude.
centerLat	Map center latitude.
maptype	Defaults to Esri Topographic ³³ Available to select between Stamen basemaps or Esri basemaps. Stamen <ul style="list-style-type: none"> • terrain • terrain-background • terrain-labels • terrain-lines • toner • toner-background • toner-hybrid • toner-labels • toner-labels • toner-labels • toner-lines • toner-lite • watercolor Esri <ul style="list-style-type: none"> • "world_topo" • "world_imagery" • "world_terrain" • "de_Lorme" • "world_grey" • "world_streets"
zoom	Map zoom level.
width	Width of image, in pixels.
height	Height of image, in pixels.
bbox	If you are using the Esri maps, then the <code>bbox</code> parameter must be an <code>st_bbox</code> object as specified in the <code>sf</code> package documentation https://www.rdocumentation.org/packages/sf/versions/0.7-4/topics/st_bbox . If using Stamen Maps, use a vector organized as (lonLo, latLo, lonHi, latHi) If not null, centerLon, centerLat, and zoom are ignored.
maxTiles	Only utilized if selecting an esri basemap, specifies the maximum number of tiles to be returned. The greater the number, the slower the performance – arbitrarily set to 20 by default.
crs	Object of class CRS. The Coordinate Reference System (CRS) for the returned map. If the CRS of the downloaded map does not match, it will be projected to the specified CRS using <code>raster::projectRaster</code> .
tileCacheDir	Optional location for cached tiles.

³³<http://www.arcgis.com/home/item.html?id=30e5fe3149c34df1ba922e6f5bbf808f>

Value

A rasterBrick object which can be plotted with `staticmap_plotRasterBrick()` or `raster::plotRGB()` and serve as a base plot.

Note

The spatial reference of the image when it is downloaded is 3857. If the crs argument is different, projecting may cause the size and extent of the image to differ very slightly from the input, on a scale of 1-2 pixels or 10^{-3} degrees.

If bbox is specified and the bbox aspect ratio does not match the width/height aspect ratio the extent is resized to prevent the map image from appearing stretched, so the map extent may not match the bbox argument exactly.

See Also

```
staticmap_getStamenmapBrick
staticmap_getEsrimapBrick
staticmap_plotRasterBrick
```

Examples

```
## Not run:
rasterBrick <- staticmap_getRasterBrick(-122.3318, 47.668)
staticmap_plotRasterBrick(rasterBrick)

## End(Not run)
## Not run:
rasterBrick <- staticmap_getRasterBrick(-122.3318, 47.668, "world_streets", 12)
staticmap_plotRasterBrick(rasterBrick)

## End(Not run)
## Not run:
rasterBrick <- staticmap_getRasterBrick(-122.3318, 47.668, "watercolor", 12)
staticmap_plotRasterBrick(rasterBrick)

## End(Not run)
```

```
staticmap_getStamenmapBrick
Create a rasterBrick from stamenmap tiles
```

Description

Downloads a PNG from the stamenmap tile server and creates a `raster::rasterBrick` object with layers for red, green, and blue. This can then be passed as the `rasterBrick` object to the `staticmap_plotRasterBrick()` function for plotting.

Stamen maps tiles are freely available (April, 2019) and are described at the following URL:

<http://maps.stamen.com/#terrain/12/37.7706/-122.3782>

"These tiles are made available as part of the CityTracking project, funded by the Knight Foundation, in which Stamen is building web services and open source tools to display public data in easy-to-understand, highly visual ways."

Usage

```
staticmap_getStamenmapBrick(centerLon = NULL, centerLat = NULL,
  maptype = "terrain", zoom = 12, width = 640, height = 640,
  bbox = NULL, crs = sp::CRS("+init=epsg:4326"),
  tileCacheDir = tempdir())
```

Arguments

centerLon	map center longitude
centerLat	map center latitude
maptype	map type
zoom	map zoom level; corresponds to <code>ggmap::get_map()</code> zoom level
width	width of image, in pixels
height	height of image, in pixels
bbox	bounding box vector (lonLo, latLo, lonHi, latHi). If not null, centerLon, centerLat, and zoom are ignored.
crs	object of class CRS. The Coordinate Reference System (CRS) for the returned map. If the CRS of the downloaded map does not match, it will be projected to the specified CRS using <code>raster::projectRaster</code> .
tileCacheDir	Optional location for cached tiles.

Value

A `rasterBrick` object which can be plotted with `staticmap_plotRasterBrick()` or `raster::plotRGB()` and serve as a base plot.

Note

The spatial reference of the image when it is downloaded is 3857. If the `crs` argument is different, projecting may cause the size and extent of the image to differ very slightly from the input, on a scale of 1-2 pixels or 10^{-3} degrees.

If `bbox` is specified and the `bbox` aspect ratio does not match the width/height aspect ratio the extent is resized to prevent the map image from appearing stretched, so the map extent may not match the `bbox` argument exactly.

See Also

`staticmap_getRasterBrick`
`staticmap_plotRasterBrick`

Examples

```
## Not run:
rasterBrick <- staticmap_getStamenmapBrick(-122.3318, 47.668)
staticmap_plotRasterBrick(rasterBrick)

## End(Not run)
```

```
staticmap_plotRasterBrick
Plot an RGB rasterBrick
```

Description

Plots the incoming rasterBrick.

Usage

```
staticmap_plotRasterBrick(rasterBrick = NULL, grayscale = FALSE, ...)
```

Arguments

`rasterBrick` an RGB rasterBrick object. It is assumed that layer 1 represents red, layer 2 represents green, and layer 3 represents blue.

`grayscale` logical specifying conversion to grayscale

`...` arguments passed on to `raster::plot()` (for `grayscale = TRUE`) or `raster::plotRGB()` (for `grayscale = FALSE`)

See Also

`staticmap_getStamenmapBrick`

Examples

```
## Not run:
rasterBrick <- staticmap_getStamenmapBrick(-122.3318, 47.668)
staticmap_plotRasterBrick(rasterBrick)
staticmap_plotRasterBrick(rasterBrick, grayscale = TRUE)

## End(Not run)
```

tidy_toMonitor	<i>Convert 'ws_tidy' data to a 'ws_monitor' object</i>
----------------	--

Description

Changes read-optimized 'tidy' formatted monitor data into a write-optimized 'ws_monitor' format. If the given data is already a 'ws_monitor' object, it is returned as is. This function is the inverse of `monitor_toTidy`.

Usage

```
tidy_toMonitor(data = NULL)
```

Arguments

`data` Data to potentially convert.

Value

'ws_monitor' object

Examples

```
ws_monitor <- monitor_subset(
  Northwest_Megafires,
  monitorIDs = c('530470009_01', '530470010_01')
)

ws_monTidy <- monitor_toTidy(ws_monitor)
ws_monMon <- tidy_toMonitor(ws_monTidy)
head(ws_monMon$data)
head(ws_monitor$data)
```

timeInfo	<i>Get time related information</i>
----------	-------------------------------------

Description

Calculate the local time at the target location, as well as sunrise, sunset and solar noon times, and create several temporal masks.

If the `timezone` is provided it will be used. Otherwise, the **MazamaSpatialUtils** package will be used to determine the timezone from `longitude` and `latitude`.

The returned dataframe will have as many rows as the length of the incoming UTC time vector and will contain the following columns:

- `localStdTime_UTC` – UTC representation of local **standard** time

- `daylightSavings` – logical mask = TRUE if daylight savings is in effect
- `localTime` – local clock time
- `sunrise` – time of sunrise on each `localTime` day
- `sunset` – time of sunset on each `localTime` day
- `solarnoon` – time of solar noon on each `localTime` day
- `day` – logical mask = TRUE between sunrise and sunset
- `morning` – logical mask = TRUE between sunrise and solarnoon
- `afternoon` – logical mask = TRUE between solarnoon and sunset
- `night` – logical mask = opposite of day

Usage

```
timeInfo(time, longitude = NULL, latitude = NULL, timezone = NULL)
```

Arguments

<code>time</code>	POSIXct vector with specified timezone,
<code>longitude</code>	Longitude of the location of interest.
<code>latitude</code>	Latitude of the location of interest.
<code>timezone</code>	Olson timezone at the location of interest.

Details

While the **lubridate** package makes it easy to work in local timezones, there is no easy way in R to work in "Local Standard Time" (LST) as is often required when working with air quality data. EPA regulations mandate that daily averages be calculated based on LST.

The `localStdTime_UTC` column in the returned dataframe is primarily for internal use and provides an important tool for creating LST daily averages and LST axis labeling.

Value

A dataframe with times and masks.

Examples

```
carmel <- monitor_subset(Carmel_Valley, tlim = c(20160801,20160810))

# Create timeInfo object for this monitor
ti <- timeInfo(carmel$data$datetime,
               carmel$meta$longitude,
               carmel$meta$latitude,
               carmel$meta$timezone)

# Subset the data based on day/night masks
data_day <- carmel$data[ti$day,]
data_night <- carmel$data[ti$night,]
```

```
# Build two monitor objects
carmel_day <- list(meta = carmel$meta, data = data_day)
carmel_night <- list(meta = carmel$meta, data = data_night)

# Plot them
monitor_timeseriesPlot(carmel_day, shadedNight = TRUE, pch = 8, col = 'goldenrod')
monitor_timeseriesPlot(carmel_night, pch = 16, col = 'darkblue', add = TRUE)
```

upgradeMeta_v1.0 *Upgrade ws_monitor Metadata to Version 1.0*

Description

Upgrade a *ws_monitor* object to version 1.0 standards.

Usage

```
upgradeMeta_v1.0(ws_monitor)
```

Arguments

ws_monitor *ws_monitor* object

Value

A *ws_monitor* object with version 1.0 metadata.

US_52 *US State Codes*

Description

State codes for the 50 states +DC +PR (Puerto Rico)

Usage

```
US_52
```

Format

A vector with 52 elements

Details

US state codes

 WRCC

WRCC Monitor Names and Unit IDs

Description

The WRCC <https://wrcc.dri.edu/cgi-bin/smoke.pl> Fire Cache Smoke Monitor Archive provides access to a variety of monitors that can be accessed with the `wrcc_createMonitorObject` function. Use of this function requires a valid `unitID`. The WRCC object is a list of lists. The element named `unitIDs` is itself a list of three named vectors, each containing the `unitIDs` and associated names for one of the categories of monitors available at WRCC:

- `cache`
- `miscellaneous`
- `usfs_regional`

Format

A list of lists

Details

WRCC monitor names and `unitIDs`

Note

This list of monitor IDs reflects `unitIDs` found on the WRCC site on June 12, 2019.

`wrcc_createDataDataframe`

Create WRCC data dataframe

Description

After quality control has been applied to an WRCC tibble, we can extract the PM2.5 values and store them in a data tibble organized as time-by-deployment (aka time-by-site).

The first column of the returned dataframe is named `'datetime'` and contains a POSIXct time in UTC. Additional columns contain data for each separate deployment of a monitor.

Usage

```
wrcc_createDataDataframe(tbl, meta)
```

Arguments

<code>tbl</code>	single site WRCC tibble created by <code>wrcc_clustering()</code>
<code>meta</code>	WRCC meta dataframe created by <code>wrcc_createMetaDataframe()</code>

Value

A data dataframe for use in a *ws_monitor* object.

```
wrcc_createMetaDataframe
      Create WRCC site location metadata dataframe
```

Description

After a WRCC tibble has been enhanced with additional columns generated by `addClustering` we are ready to pull out site information associated with unique deployments.

These will be rearranged into a dataframe organized as deployment-by-property with one row for each monitor deployment.

This site information found in `tbl` is augmented so that we end up with a uniform set of properties associated with each monitor deployment. The list of columns in the returned `meta` dataframe is:

```
> names(p$meta)
 [1] "monitorID"           "longitude"           "latitude"
 [4] "elevation"           "timezone"            "countryCode"
 [7] "stateCode"           "siteName"            "agencyName"
[10] "countyName"          "msaName"             "monitorType"
[13] "monitorInstrument"   "aqslID"              "pwfslID"
[16] "pwfslDataIngestSource" "telemetryAggregator" "telemetryUnitID"
```

Usage

```
wrcc_createMetaDataframe(tbl, unitID = as.character(NA),
  pwfslDataIngestSource = "WRCC", existingMeta = NULL,
  addGoogleMeta = FALSE, addEsriMeta = FALSE)
```

Arguments

```
tbl           single site WRCC tibble after metadata enhancement
unitID        character or numeric WRCC unit identifier
pwfslDataIngestSource
              identifier for the source of monitoring data, e.g. 'WRCC '
existingMeta   existing 'meta' dataframe from which to obtain metadata for known monitor
              deployments
addGoogleMeta logical specifying wheter to use Google elevation and reverse geocoding ser-
              vices
addEsriMeta   logical specifying wheter to use ESRI elevation and reverse geocoding services
```

Value

A `meta` dataframe for use in a *ws_monitor* object.

See Also

addMazamaMetadata

wrcc_createMonitorObject

Obtain WRCC data and create ws_monitor object

Description

Obtains monitor data from an WRCC webservice and converts it into a quality controlled, metadata enhanced *ws_monitor* object ready for use with all *monitor_~* functions.

Steps involved include:

1. download CSV text
2. parse CSV text
3. apply quality control
4. apply clustering to determine unique deployments
5. enhance metadata to include: elevation, timezone, state, country, site name
6. reshape data into deployment-by-property *meta* and and time-by-deployment *data* dataframes

QC parameters that can be passed in the ... include the following valid data ranges as taken from `wrcc_EBAMQualityControl()`:

- `valid_Longitude=c(-180,180)`
- `valid_Latitude=c(-90,90)`
- `remove_Lon_zero = TRUE`
- `remove_Lat_zero = TRUE`
- `valid_Flow = c(16.7*0.95,16.7*1.05)`
- `valid_AT = c(-Inf,45)`
- `valid_RHi = c(-Inf,45)`
- `valid_Conc = c(-Inf,5000)`

Note that appropriate values for QC thresholds will depend on the type of monitor.

Usage

```
wrcc_createMonitorObject(startdate = strftime(lubridate::now(),
"%Y010100", tz = "UTC"), enddate = strftime(lubridate::now(),
"%Y%m%d23", tz = "UTC"), unitID = NULL, clusterDiameter = 1000,
zeroMinimum = TRUE,
baseUrl = "https://wrcc.dri.edu/cgi-bin/wea_list2.pl",
saveFile = NULL, existingMeta = NULL, addGoogleMeta = FALSE,
addEsriMeta = FALSE, ...)
```

Arguments

startdate	desired start date (integer or character representing YYYYMMDD[HH])
enddate	desired end date (integer or character representing YYYYMMDD[HH])
unitID	station identifier (will be upcased)
clusterDiameter	diameter in meters used to determine the number of clusters (see addClustering)
zeroMinimum	logical specifying whether to convert negative values to zero
baseUrl	base URL for data queries
saveFile	optional filename where raw CSV will be written
existingMeta	existing 'meta' dataframe from which to obtain metadata for known monitor deployments
addGoogleMeta	logical specifying wheter to use Google elevation and reverse geocoding services
addEsriMeta	logical specifying wheter to use ESRI elevation and reverse geocoding services
...	additional parameters are passed to type-specific QC functions

Value

A *ws_monitor* object with WRCC data.

Note

The downloaded CSV may be saved to a local file by providing an argument to the `saveFile` parameter.

See Also

```
wrcc_downloadData
wrcc_parseData
wrcc_qualityControl
addClustering
wrcc_createMetaDataframe
wrcc_createDataDataframe
```

Examples

```
## Not run:
initializeMazamaSpatialUtils()
sm13 <- wrcc_createMonitorObject(20150301, 20150831, unitID = 'sm13')
monitor_leaflet(sm13)

## End(Not run)
```

```
wrcc_createRawDataframe
```

Obtain WRCC data and parse into a tibble

Description

Obtains monitor data from a WRCC webservice and converts it into a quality controlled, metadata enhanced "raw" tibble ready for use with all `raw_~` functions.

Steps involved include:

1. download CSV text
2. parse CSV text
3. apply quality control
4. apply clustering to determine unique deployments
5. enhance metadata to include: elevation, timezone, state, country, site name

Usage

```
wrcc_createRawDataframe(startdate = strftime(lubridate::now(),
"%Y010100", tz = "UTC"), enddate = strftime(lubridate::now(),
"%Y%m%d23", tz = "UTC"), unitID = NULL, clusterDiameter = 1000,
baseUrl = "https://wrcc.dri.edu/cgi-bin/wea_list2.pl",
saveFile = NULL, flagAndKeep = FALSE)
```

Arguments

<code>startdate</code>	Desired start date (integer or character representing YYYYMMDD[HH]).
<code>enddate</code>	Desired end date (integer or character representing YYYYMMDD[HH]).
<code>unitID</code>	Station identifier (will be upcased).
<code>clusterDiameter</code>	Diameter in meters used to determine the number of clusters (see <code>addClustering</code>).
<code>baseUrl</code>	Base URL for data queries.
<code>saveFile</code>	Optional filename where raw CSV will be written.
<code>flagAndKeep</code>	Flag, rather than remove, bad data during the QC process.

Value

Raw tibble of WRCC data.

Note

The downloaded CSV may be saved to a local file by providing an argument to the `saveFile` parameter.

Monitor unitIDs can be found at <https://wrcc.dri.edu/cgi-bin/smoke.pl>.

References

Fire Cache Smoke Monitoring Archive³⁴

See Also

wrcc_downloadData
 wrcc_parseData
 wrcc_qualityControl
 addClustering

Examples

```
## Not run:
tbl <- wrcc_createRawDataframe(20150701, 20150930, unitID = 'SM16')

## End(Not run)
```

wrcc_downloadData *Download WRCC data*

Description

Request data from a particular station for the desired time period. Data are returned as a single character string containing the WRCC output.

Monitor unitIDs can be found at <https://wrcc.dri.edu/cgi-bin/smoke.pl>.

Usage

```
wrcc_downloadData(startdate = strftime(lubridate::now(), "%Y010101", tz = "UTC"),
  enddate = strftime(lubridate::now(), "%Y%m%d23", tz = "UTC"),
  unitID = NULL,
  baseUrl = "https://wrcc.dri.edu/cgi-bin/wea_list2.pl")
```

Arguments

startdate	desired start date (integer or character representing YYYYMMDD[HH])
enddate	desired end date (integer or character representing YYYYMMDD[HH])
unitID	station identifier (will be upcased)
baseUrl	base URL for data queries

Value

String containing WRCC output.

³⁴<https://wrcc.dri.edu/cgi-bin/smoke.pl>

References

Fire Cache Smoke Monitoring Archive³⁵

Examples

```
## Not run:
fileString <- wrcc_downloadData(20150701, 20150930, unitID = 'SM16')
df <- wrcc_parseData(fileString)

## End(Not run)
```

```
wrcc_EBAMQualityControl
```

Apply Quality Control to raw WRCC EBAM tibble

Description

Perform various QC measures on WRCC EBAM data.

The any numeric values matching the following are converted to NA

- $x < -900$
- $x == -9.9899$
- $x == 99999$

The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on DateTime.

Usage

```
wrcc_EBAMQualityControl(tbl, valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90), remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE, valid_Flow = c(16.7 * 0.95, 16.7 * 1.05),
  valid_AT = c(-Inf, 45), valid_RHi = c(-Inf, 45),
  valid_Conc = c(-Inf, 5000), flagAndKeep = FALSE)
```

³⁵<https://wrcc.dri.edu/cgi-bin/smoke.pl>

Arguments

tbl single site tibble created by wrcc_parseData ()
 valid_Longitude range of valid Longitude values
 valid_Latitude range of valid Latitude values
 remove_Lon_zero flag to remove rows where Longitude == 0
 remove_Lat_zero flag to remove rows where Latitude == 0
 valid_Flow range of valid Flow values
 valid_AT range of valid AT values
 valid_RHi range of valid RHi values
 valid_Conc range of valid ConcHr values
 flagAndKeep flag, rather than remove, bad data during the QC process

Value

Cleaned up tibble of WRCC monitor data.

See Also

wrcc_qualityControl

wrcc_ESAMQualityControl

Apply Quality Control to raw WRCC E-Sampler tibble

Description

Perform various QC measures on WRCC EBAM data.

The any numeric values matching the following are converted to NA

- $x < -900$
- $x == -9.9899$
- $x == 99999$

The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on DateTime.

Usage

```
wrcc_ESAMQualityControl(tbl, valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90), remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE, valid_Flow = c(1.999, 2.001),
  valid_AT = c(-Inf, 150), valid_RHi = c(-Inf, 55),
  valid_Conc = c(-Inf, 5000), flagAndKeep = FALSE)
```

Arguments

tbl	single site tibble created by wrcc_parseData()
valid_Longitude	range of valid Longitude values
valid_Latitude	range of valid Latitude values
remove_Lon_zero	flag to remove rows where Longitude == 0
remove_Lat_zero	flag to remove rows where Latitude == 0
valid_Flow	range of valid Flow values
valid_AT	range of valid AT values
valid_RHi	range of valid RHi values
valid_Conc	range of valid ConcHr values
flagAndKeep	flag, rather than remove, bad data during the QC process

Value

Cleaned up tibble of WRCC monitor data.

See Also

wrcc_qualityControl

wrcc_identifyMonitorType
Identify WRCC monitor type

Description

Examine the column names of the incoming character vector to identify different types of monitor data provided by WRCC.

The return is a list includes everything needed to identify and parse the raw data using readr::read_tsv():

- monitorType – identification string
- rawNames – column names from the data (including special characters)

- columnNames – assigned column names (special characters repaced with '.')
- columnTypes – column type string for use with readr::read_csv()

The monitorType will be one of:

- "WRCC_TYPE1" – ???
- "WRCC_TYPE2" – ???
- "UNKOWN" – ???

Usage

```
wrcc_identifyMonitorType(fileString)
```

Arguments

fileString character string containing WRCC data

Value

List including monitorType, rawNames, columnNames and columnTypes.

References

WRCC Fire Cache Smoke Monitor Archive³⁶

Examples

```
## Not run:
fileString <- wrcc_downloadData(20160701, 20160930, unitID='1307')
monitorTypeList <- wrcc_identifyMonitorType(fileString)

## End(Not run)
```

wrcc_load

Load Processed WRCC Monitoring Data

Description

Please use wrcc_loadAnnual instead of this function. It will soon be deprecated.

Usage

```
wrcc_load(year = 2017,
          baseUrl = "https://haze.airfire.org/monitoring/WRCC/RData/")
```

³⁶<https://wrcc.dri.edu/cgi-bin/smoke.pl>

Arguments

year desired year (integer or character representing YYYY)
 baseUrl base URL for WRCC meta and data files

Value

A *ws_monitor* object with WRCC data.

wrcc_loadAnnual *Load annual WRCC monitoring data*

Description

Loads pre-generated .RData files containing annual WRCC data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The annual files loaded by this function are updated on the 15th of each month and cover the period from the beginning of the year to the end of the last month.

For data during the last 45 days, use `wrcc_loadDaily()`.

For the most recent data, use `wrcc_loadLatest()`.

WRCC parameters include the following:

1. PM2.5

Available WRCC RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/WRCC/RData>³⁷

Usage

```
wrcc_loadAnnual(year = NULL, parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring", dataDir = NULL)
```

Arguments

year Desired year (integer or character representing YYYY).
 parameter Parameter of interest.
 baseUrl Base URL for 'annual' WRCC data files.
 dataDir Local directory containing 'annual' data files.

Value

A *ws_monitor* object with WRCC data.

³⁷<https://haze.airfire.org/monitoring/WRCC/RData>

See Also

wrcc_loadDaily
wrcc_loadLatest

Examples

```
## Not run:
wrcc_loadAnnual(2017) %>%
  monitor_subset(stateCodes='MT', tlim=c(20170701,20170930)) %>%
  monitor_dailyStatistic() %>%
  monitor_timeseriesPlot(style = 'gnats', ylim=c(0,300), xpd=NA)
  addAQIStackedBar()
  addAQILines()
  title("Montana 2017 -- WRCC Daily Average PM2.5")

## End(Not run)
```

wrcc_loadDaily *Load recent WRCC monitoring data*

Description

Loads pre-generated .RData files containing recent WRCC data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The daily files loaded by this function are updated once a day, shortly after midnight and contain data for the previous 45 days.

For the most recent data, use `wrcc_loadLatest()`.

For data extended more than 45 days into the past, use `wrcc_loadAnnual()`.

WRCC parameters include the following:

1. PM2.5

Avaiable WRCC RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/WRCC/RData/latest>³⁸

Usage

```
wrcc_loadDaily(parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL)
```

Arguments

<code>parameter</code>	Parameter of interest.
<code>baseUrl</code>	Base URL for 'daily' AirNow data files.
<code>dataDir</code>	Local directory containing 'daily' data files.

³⁸<https://haze.airfire.org/monitoring/WRCC/RData/latest>

Value

A *ws_monitor* object with WRCC data.

See Also

wrcc_loadAnnual
wrcc_loadLatest

Examples

```
## Not run:
wrcc_loadDaily() %>%
  monitor_subset(stateCodes=CONUS) %>%
  monitor_map()

## End(Not run)
```

wrcc_loadLatest	<i>Load most recent WRCC monitoring data</i>
-----------------	--

Description

Loads pre-generated .RData files containing the most recent WRCC data.

If *dataDir* is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The files loaded by this function are updated multiple times an hour and contain data for the previous 10 days.

For daily updates covering the most recent 45 days, use `wrcc_loadDaily()`.

For data extended more than 45 days into the past, use `wrcc_loadAnnual()`.

WRCC parameters include the following:

1. PM2.5

Avaiable RData and associated log files can be seen at: <https://haze.airfire.org/monitoring/WRCC/RData/latest>³⁹

Usage

```
wrcc_loadLatest(parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL)
```

Arguments

<i>parameter</i>	Parameter of interest.
<i>baseUrl</i>	Base URL for 'daily' AirNow data files.
<i>dataDir</i>	Local directory containing 'daily' data files.

³⁹<https://haze.airfire.org/monitoring/WRCC/RData/latest>

Value

A *ws_monitor* object with WRCC data.

See Also

wrcc_loadAnnual
wrcc_loadDaily

Examples

```
## Not run:
wrcc_loadLatest() %>%
  monitor_subset(stateCodes=CONUS) %>%
  monitor_map()

## End(Not run)
```

wrcc_parseData	<i>Parse WRCC data string</i>
----------------	-------------------------------

Description

Raw character data from WRCC are parsed into a tibble. The incoming `fileString` can be read in directly from WRCC using `wrcc_downloadData()` or from a local file using `readr::read_file()`.

The type of monitor represented by this `fileString` is inferred from the column names using `wrcc_identifyMonitorType` and appropriate column types are assigned. The character data are then processed, read into a tibble and augmented in the following ways:

1. Spaces at the beginning and end of each line are moved.
2. All header lines beginning with ':' are removed.

Usage

```
wrcc_parseData(fileString)
```

Arguments

`fileString` character string containing WRCC data

Value

Dataframe of WRCC raw monitor data.

References

Fire Cache Smoke Monitoring Archive⁴⁰

⁴⁰<https://wrcc.dri.edu/cgi-bin/smoke.pl>

Examples

```
## Not run:
fileString <- wrcc_downloadData(20150701, 20150930, unitID = 'SM16')
tbl <- wrcc_parseData(fileString)

## End(Not run)
```

```
wrcc_qualityControl
```

Apply Quality Control to raw WRCC tibble

Description

Various QC steps are taken to clean up the incoming raw tibble including:

1. Convert numeric missing value flags to NA.
2. Remove measurement records with values outside of valid ranges.

See the individual `wrcc_~QualityControl()` functions for details.

Usage

```
wrcc_qualityControl(tbl, ...)
```

Arguments

<code>tbl</code>	single site tibble created by <code>wrcc_downloadData()</code>
<code>...</code>	additional parameters are passed to type-specific QC functions

Value

Cleaned up tibble of WRCC monitor data.

See Also

```
wrcc_EBAMQualityControl
wrcc_ESAMQualityControl
```