

# Package: gdalcubes (via r-universe)

October 26, 2024

**Title** Earth Observation Data Cubes from Satellite Image Collections

**Version** 0.7.0

**Date** 2024-03-06

**Description** Processing collections of Earth observation images as on-demand multispectral, multitemporal raster data cubes. Users define cubes by spatiotemporal extent, resolution, and spatial reference system and let 'gdalcubes' automatically apply cropping, reprojection, and resampling using the 'Geospatial Data Abstraction Library' ('GDAL'). Implemented functions on data cubes include reduction over space and time, applying arithmetic expressions on pixel band values, moving window aggregates over time, filtering by space, time, bands, and predicates on pixel values, exporting data cubes as 'netCDF' or 'GeoTIFF' files, plotting, and extraction from spatial and or spatiotemporal features. All computational parts are implemented in C++, linking to the 'GDAL', 'netCDF', 'CURL', and 'SQLite' libraries. See Appel and Pebesma (2019) <[doi:10.3390/data4030092](https://doi.org/10.3390/data4030092)> for further details.

**Depends** R (>= 3.4)

**Imports** Rcpp, jsonlite, ncd4

**License** MIT + file LICENSE

**URL** <https://github.com/appelmar/gdalcubes>

**BugReports** <https://github.com/appelmar/gdalcubes/issues/>

**Encoding** UTF-8

**RoxygenNote** 7.2.3

**LinkingTo** Rcpp, BH

**Suggests** knitr, rmarkdown, stars, av, gifski, sf, tinytest, lubridate

**VignetteBuilder** knitr

**Copyright** file inst/COPYRIGHTS

**NeedsCompilation** yes

**SystemRequirements** GDAL (>= 2.0.1), PROJ (>= 4.8.0), netcdf, sqlite3

**Author** Marius Appel [aut, cre]

(<<https://orcid.org/0000-0001-5281-3896>>), Edzer Pebesma [ctb]

(<<https://orcid.org/0000-0001-8049-7069>>), Roger Bivand [ctb],

Jeroen Ooms [ctb] (<<https://orcid.org/0000-0002-4035-0289>>),

Lewis Van Winkle [cph], Ole Christian Eidheim [cph], Howard

Hinnant [cph], Adrian Colomitchi [cph], Florian Dang [cph],

Paul Thompson [cph], Tomasz Kamiński [cph], Dropbox, Inc. [cph]

**Maintainer** Marius Appel <marius.appel@hs-bochum.de>

**Repository** CRAN

**Date/Publication** 2024-03-07 00:00:02 UTC

## Contents

.copy_cube . . . . .	4
add_collection_format . . . . .	4
add_images . . . . .	5
aggregate_space . . . . .	6
aggregate_time . . . . .	7
animate . . . . .	8
apply_pixel . . . . .	10
apply_pixel.array . . . . .	11
apply_pixel.cube . . . . .	12
apply_time . . . . .	14
apply_time.array . . . . .	15
apply_time.cube . . . . .	16
as.data.frame.cube . . . . .	18
as_array . . . . .	19
as_json . . . . .	20
bands . . . . .	21
chunk_apply . . . . .	21
collection_formats . . . . .	23
create_image_collection . . . . .	23
crop . . . . .	25
cube_view . . . . .	27
dim.cube . . . . .	29
dimensions . . . . .	30
dimension_bounds . . . . .	31
dimension_values . . . . .	31
extent . . . . .	32
extract_geom . . . . .	33
fill_time . . . . .	35
filter_geom . . . . .	36
filter_pixel . . . . .	38
gdalcubes . . . . .	39
gdalcubes_gdalformats . . . . .	39

gdalcubes_gdalversion . . . . .	40
gdalcubes_gdal_has_geos . . . . .	40
gdalcubes_options . . . . .	40
gdalcubes_selection . . . . .	42
gdalcubes_set_gdal_config . . . . .	44
image_collection . . . . .	45
image_mask . . . . .	45
join_bands . . . . .	47
json_cube . . . . .	48
memsize . . . . .	49
names.cube . . . . .	50
nbands . . . . .	50
ncdf_cube . . . . .	51
nt . . . . .	52
nx . . . . .	53
ny . . . . .	54
pack_minmax . . . . .	54
plot.cube . . . . .	55
predict.cube . . . . .	58
print.cube . . . . .	59
print.cube_view . . . . .	60
print.image_collection . . . . .	61
proj4 . . . . .	61
raster_cube . . . . .	62
read_chunk_as_array . . . . .	63
reduce_space . . . . .	65
reduce_space.array . . . . .	66
reduce_space.cube . . . . .	67
reduce_time . . . . .	68
reduce_time.array . . . . .	69
reduce_time.cube . . . . .	70
rename_bands . . . . .	72
select_bands . . . . .	73
select_time . . . . .	74
size . . . . .	75
slice_space . . . . .	76
slice_time . . . . .	77
srs . . . . .	78
stack_cube . . . . .	79
stac_image_collection . . . . .	81
st_as_stars.cube . . . . .	82
window_space . . . . .	83
window_time . . . . .	85
write_chunk_from_array . . . . .	86
write_ncdf . . . . .	87
write_tif . . . . .	89

---

<code>.copy_cube</code>	<i>Create a data cube proxy object copy</i>
-------------------------	---

---

**Description**

Copy a data cube proxy object without copying any data

**Usage**

```
.copy_cube(cube)
```

**Arguments**

cube	source data cube proxy object
------	-------------------------------

**Details**

This internal function copies the complete processing chain / graph of a data cube but does not copy any data. It is used internally to avoid in-place modification for operations with potential side effects on source data cubes.

**Value**

copied data cube proxy object

---

<code>add_collection_format</code>	<i>Download and install an image collection format from a URL</i>
------------------------------------	---

---

**Description**

Download and install an image collection format from a URL

**Usage**

```
add_collection_format(url, name = NULL)
```

**Arguments**

url	URL pointing to the collection format JSON file
name	optional name used to refer to the collection format

**Details**

By default, the collection format name will be derived from the basename of the URL.

**Examples**

```
add_collection_format(
    "https://raw.githubusercontent.com/appelmar/gdalcubes_cpp/dev/formats/Sentinel1_IW_GRD.json")
```

---

 add\_images

*Add images to an existing image collection*


---

**Description**

This function adds provided files or GDAL dataset identifiers and to an existing image collection by extracting datetime, image identifiers, and band information according to the collection's format.

**Usage**

```
add_images(
    image_collection,
    files,
    unroll_archives = TRUE,
    out_file = "",
    quiet = FALSE
)
```

**Arguments**

image_collection	image_collection object or path to an existing collection file
files	character vector with paths to image files on disk or any GDAL dataset identifiers (including virtual file systems and higher level drivers or GDAL sub-datasets)
unroll_archives	automatically convert .zip, .tar archives and .gz compressed files to GDAL virtual file system dataset identifiers (e.g. by prepending /vsizip/) and add contained files to the list of considered files
out_file	path to output file, an empty string (the default) will update the collection in-place, whereas images will be added to a new copy of the image collection at the given location otherwise.
quiet	logical; if TRUE, do not print resulting image collection if return value is not assigned to a variable

**Value**

image collection proxy object, which can be used to create a data cube using [raster\\_cube](#)

**Examples**

```
L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                      ".TIF", recursive = TRUE, full.names = TRUE)
L8_col = create_image_collection(L8_files[1:12], "L8_L1TP")
add_images(L8_col, L8_files[13:24])
```

---

aggregate_space	<i>Spatial aggregation of data cubes</i>
-----------------	--

---

**Description**

Create a proxy data cube, which applies an aggregation function to reduce the spatial resolution.

**Usage**

```
aggregate_space(cube, dx, dy, method = "mean", fact = NULL)
```

**Arguments**

cube	source data cube
dx	numeric value; new spatial resolution in x direction
dy	numeric value; new spatial resolution in y direction
method	aggregation method, one of "mean", "min", "max", "median", "count", "sum", "prod", "var", and "sd"
fact	simple integer factor defining how many cells (per axis) become aggregated to a single new cell, can be used instead of dx and dy

**Details**

This function reduces the spatial resolution of a data cube by applying an aggregation function to smaller blocks of pixels.

The size of the cube may be expanded automatically in all directions if the original extent is not divisible by the new size of pixels.

Notice that if boundaries of the target cube do not align with the boundaries of the input cube (for example, if aggregating from 10m to 15m spatial resolution), pixels of the input cube will contribute to the output pixel that contains its center coordinate. If the center coordinate is exactly on a boundary, the input pixel will contribute to the right / bottom pixel of the output cube.

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}
L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
  srs="EPSG:32618", dx = 500, dy=500, dt="P3M", aggregation = "median")
L8.cube = raster_cube(L8.col, v, mask=image_mask("BQA", bits=4, values=16))
L8.rgb = select_bands(L8.cube, c("B02", "B03", "B04"))
L8.5km = aggregate_space(L8.rgb, 5000,5000, "mean")
L8.5km

plot(L8.5km, rgb=3:1, zlim=c(5000,12000))
```

---

 aggregate\_time

*Aggregate data cube time series to lower temporal resolution*


---

**Description**

Create a proxy data cube, which applies an aggregation function over pixel time series to lower temporal resolution.

**Usage**

```
aggregate_time(cube, dt, method = "mean", fact = NULL)
```

**Arguments**

cube	source data cube
dt	character; new temporal resolution, datetime period string, e.g. "P1M"
method	aggregation method, one of "mean", "min", "max", "median", "count", "sum", "prod", "var", and "sd"
fact	simple integer factor defining how many cells become aggregated to a single new cell, can be used instead of dt

**Details**

This function can be used to aggregate time series to lower resolution or to regularize a data cube with irregular (labeled) time axis. It is possible to change the unit of the temporal resolution (e.g. to create monthly composites from daily images). The size of the cube may be expanded automatically if the original temporal extent is not divisible by the new temporal size of pixels.

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}
L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P3M", aggregation = "median")
L8.cube = raster_cube(L8.col, v, mask=image_mask("BQA", bits=4, values=16))
L8.rgb = select_bands(L8.cube, c("B02", "B03", "B04"))
L8.two_monthly = aggregate_time(L8.rgb, "P6M", "min")
L8.two_monthly

plot(L8.two_monthly, rgb=3:1, zlim=c(5000,12000))
```

---

 animate

*Animate a data cube as an image time series*


---

**Description**

This function can animate data cube time series as mp4 videos or animated GIFs. Depending on the desired output format, either the `av` or the `gifski` package is needed to create mp4 and GIF animations respectively.

**Usage**

```
animate(
  x,
  ...,
  fps = 1,
  loop = TRUE,
  width = 800,
  height = 800,
  save_as = tempfile(fileext = ".gif"),
  preview = interactive()
)
```



**Arguments**

x	a data cube proxy object (class cube)
...	parameters passed to plot.cube
fps	frames per second of the animation
loop	how many iterations, TRUE = infinite
width	width (in pixels) of the animation
height	height (in pixels) of the animation
save_as	character path where the animation shall be stored, must end with ".mp4" or ".gif"
preview	logical; preview the animation

**Details**

Animations can be created for single band data cubes or RGB plots of multi-band data cubes (by providing the argument `rgb`) only.

**Value**

character; path pointing to the the created file

**See Also**

[plot.cube](#)

**Examples**

```
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8-L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                        bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P16D")

animate(select_bands(raster_cube(L8.col, v), c("B02", "B03", "B04")), rgb=3:1,
        zlim=c(0,20000), fps=1, loop=1)

animate(select_bands(raster_cube(L8.col, v), c("B05")), col=terrain.colors, key.pos=1)
```

---

apply_pixel	<i>Apply a function over (multi-band) pixels</i>
-------------	--

---

**Description**

This generic function applies a function on pixels of a data cube, an R array, or other classes if implemented.

**Usage**

```
apply_pixel(x, ...)
```

**Arguments**

x	input data
...	additional arguments passed to method implementations

**Value**

return value and type depend on the class of x

**See Also**

[apply\\_pixel.cube](#)  
[apply\\_pixel.array](#)

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")

L8.col = image_collection(file.path(tempdir(), "L8.db"))
apply_pixel(raster_cube(L8.col, v), "(B05-B04)/(B05+B04)", "NDVI")

d <- c(4,16,128,128)
x <- array(rnorm(prod(d)), d)
y <- apply_pixel(x, function(v) {
  v[1] + v[2] + v[3] - v[4]
})
```

```
})
```

---

apply_pixel.array	<i>Apply a function over pixels in a four-dimensional (band, time, y, x) array</i>
-------------------	--

---

### Description

Apply a function over pixels in a four-dimensional (band, time, y, x) array

### Usage

```
## S3 method for class 'array'  
apply_pixel(x, FUN, ...)
```

### Arguments

x	four-dimensional input array with dimensions band, time, y, x (in this order)
FUN	function that receives a vector of band values in a one-dimensional array
...	further arguments passed to FUN

### Details

FUN is expected to produce a numeric vector (or scalar) where elements are interpreted as new bands in the result.

### Note

This is a helper function that uses the same dimension ordering as gdalcubes. It can be used to simplify the application of R functions e.g. over time series in a data cube.

### Examples

```
d <- c(4, 16, 32, 32)  
x <- array(rnorm(prod(d)), d)  
y <- apply_pixel(x, function(v) {  
  v[1] + v[2] + v[3] - v[4]  
})  
dim(y)
```

---

apply\_pixel.cube      *Apply arithmetic expressions over all pixels of a data cube*

---

### Description

Create a proxy data cube, which applies arithmetic expressions over all pixels of a data cube. Expressions may access band values by name.

### Usage

```
## S3 method for class 'cube'
apply_pixel(
  x,
  expr,
  names = NULL,
  keep_bands = FALSE,
  ...,
  FUN,
  load_pkgs = FALSE,
  load_env = FALSE
)
```

### Arguments

x	source data cube
expr	character vector with one or more arithmetic expressions (see Details)
names	optional character vector with the same length as expr to specify band names for the output cube
keep_bands	logical; keep bands of input data cube, defaults to FALSE, i.e. original bands will be dropped
...	not used
FUN	user-defined R function that is applied on all pixels (see Details)
load_pkgs	logical or character; if TRUE, all currently attached packages will be attached automatically before executing FUN in spawned R processes, specific packages can alternatively be provided as a character vector.
load_env	logical or environment; if TRUE, the current global environment will be restored automatically before executing FUN in spawned R processes, can be set to a custom environment.

### Details

The function can either apply simple arithmetic C expressions given as a character vector (expr argument), or apply a custom R reducer function if FUN is provided.

In the former case, gdalcubes uses the [tinyexpr library](#) to evaluate expressions in C / C++, you can look at the [library documentation](#) to see what kind of expressions you can execute. Pixel band

values can be accessed by name. Predefined variables that can be used within the expression include integer pixel indexes (ix, iy, it), and pixel coordinates (left, right, top, bottom), t0, t1), where the last two values are provided seconds since epoch time.

FUN receives values of the bands from one pixel as a (named) vector and should return a numeric vector with identical length for all pixels. Elements of the result vectors will be interpreted as bands in the result data cube. Notice that by default, since FUN is executed in a separate R process, it cannot access any variables from outside and required packages must be loaded within FUN. To restore the current environment and automatically load packages, set load\_env and/or load\_pkgs to TRUE.

For more details and examples on how to write user-defined functions, please refer to the gdalcubes website at <https://gdalcubes.github.io/source/concepts/udfs.html>.

### Value

a proxy data cube object

### Note

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

# 1. Apply a C expression
L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
L8.cube = raster_cube(L8.col, v)
L8.cube = select_bands(L8.cube, c("B04", "B05"))
L8.ndvi = apply_pixel(L8.cube, "(B05-B04)/(B05+B04)", "NDVI")
L8.ndvi

plot(L8.ndvi)

# 2. Apply a user defined R function
L8.ndvi.noisy = apply_pixel(L8.cube, names="NDVI_noisy",
  FUN=function(x) {
    rnorm(1, 0, 0.1) + (x["B05"]-x["B04"])/(x["B05"]+x["B04"])
  })
L8.ndvi.noisy
```

```
plot(L8.ndvi.noisy)
```

---

apply_time	<i>Apply a function over (multi-band) pixel time series</i>
------------	---

---

### Description

This generic function applies a function on pixel time series of a data cube, an R array, or other classes if implemented. The resulting object is expected to have the same spatial and temporal shape as the input, i.e., no reduction is performed.

### Usage

```
apply_time(x, ...)
```

### Arguments

x	input data
...	additional arguments passed to method implementations

### Value

return value and type depend on the class of x

### See Also

[apply\\_time.cube](#)  
[apply\\_time.array](#)

### Examples

```
# 1. input is data cube
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-01", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
```

```

L8.cube = raster_cube(L8.col, v)
L8.cube = select_bands(L8.cube, c("B04", "B05"))
L8.ndvi = apply_pixel(L8.cube, "(B05-B04)/(B05+B04)", "NDVI")

# Apply a user defined R function
apply_time(L8.ndvi, names="NDVI_residuals",
  FUN=function(x) {
    y = x["NDVI",]
    if (sum(is.finite(y)) < 3) {
      return(rep(NA,ncol(x)))
    }
    t = 1:ncol(x)
    return(predict(lm(y ~ t)) - x["NDVI",]))

# 2. input is array
d <- c(4,16,32,32)
x <- array(rnorm(prod(d)), d)
z <- apply_time(x, function(v) {
  y = matrix(NA, ncol=ncol(v), nrow=2)
  y[1,] = (v[1,] + v[2,]) / 2
  y[2,] = (v[3,] + v[4,]) / 2
  y
})
dim(z)

```

---

apply_time.array	<i>Apply a function over pixel time series in a four-dimensional (band, time, y, x) array</i>
------------------	---

---

## Description

Apply a function over pixel time series in a four-dimensional (band, time, y, x) array

## Usage

```
## S3 method for class 'array'
apply_time(x, FUN, ...)
```

## Arguments

x	four-dimensional input array with dimensions band, time, y, x (in this order)
FUN	function that receives a vector of band values in a one-dimensional array
...	further arguments passed to FUN

## Details

FUN is expected to produce a matrix (or vector if result has only one band) where rows are interpreted as new bands and columns represent time.

**Note**

This is a helper function that uses the same dimension ordering as `gdalcubes`. It can be used to simplify the application of R functions e.g. over time series in a data cube.

**Examples**

```
d <- c(4,16,32,32)
x <- array(rnorm(prod(d)), d)
z <- apply_time(x, function(v) {
  y = matrix(NA, ncol=ncol(v), nrow=2)
  y[1,] = (v[1,] + v[2,]) / 2
  y[2,] = (v[3,] + v[4,]) / 2
  y
})
dim(z)
```

---

apply\_time.cube

*Apply a user-defined R function over (multi-band) pixel time series*

---

**Description**

Create a proxy data cube, which applies a user-defined R function over all pixel time series of a data cube. In contrast to `reduce_time`, the time dimension is not reduced, i.e., resulting time series must have identical length as the input data cube but may contain a different number of bands / variables. Example uses of this function may include time series decompositions, cumulative sums / products, smoothing, sophisticated NA filling, or similar.

**Usage**

```
## S3 method for class 'cube'
apply_time(
  x,
  names = NULL,
  keep_bands = FALSE,
  FUN,
  load_pkgs = FALSE,
  load_env = FALSE,
  ...
)
```

**Arguments**

<code>x</code>	source data cube
<code>names</code>	optional character vector to specify band names for the output cube
<code>keep_bands</code>	logical; keep bands of input data cube, defaults to FALSE, i.e., original bands will be dropped
<code>FUN</code>	user-defined R function that is applied on all pixel time series (see Details)



load_pkgs	logical or character; if TRUE, all currently attached packages will be attached automatically before executing FUN in spawned R processes, specific packages can alternatively be provided as a character vector.
load_env	logical or environment; if TRUE, the current global environment will be restored automatically before executing FUN in spawned R processes, can be set to a custom environment.
...	not used

### Details

FUN receives a single (multi-band) pixel time series as a matrix with rows corresponding to bands and columns corresponding to time. In general, the function must return a matrix with the same number of columns. If the result contains only a single band, it may alternatively return a vector with length identical to the length of the input time series (number of columns of the input).

For more details and examples on how to write user-defined functions, please refer to the gdalcubes website at <https://gdalcubes.github.io/source/concepts/udfs.html>.

### Value

a proxy data cube object

### Note

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-06"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
L8.cube = raster_cube(L8.col, v)
L8.cube = select_bands(L8.cube, c("B04", "B05"))
L8.ndvi = apply_pixel(L8.cube, "(B05-B04)/(B05+B04)", "NDVI")

# Apply a user defined R function
L8.ndvi.resid = apply_time(L8.ndvi, names="NDVI_residuals",
  FUN=function(x) {
    y = x["NDVI",]
    if (sum(is.finite(y)) < 3) {
      return(rep(NA, ncol(x)))
    }
  })
```

```

    }
    t = 1:ncol(x)
    return(predict(lm(y ~ t)) - x["NDVI",])
  })
L8.ndvi.resid

```

```
plot(L8.ndvi.resid)
```

---

as.data.frame.cube      *Convert a data cube to a data.frame*

---

## Description

Convert a data cube to a data.frame

## Usage

```
## S3 method for class 'cube'
as.data.frame(x, ..., complete_only = FALSE)
```

## Arguments

x	data cube object
...	not used
complete_only	logical; if TRUE, remove rows with one or more missing values

## Value

A data.frame with bands / variables of the cube as columns and pixels as rows

## Examples

```

# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8-L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-05"),
              srs="EPSG:32618", nx = 100, ny=100, dt="P1M")
x = select_bands(raster_cube(L8.col, v), c("B04", "B05"))
df = as.data.frame(x, complete_only = TRUE)

```

```
head(df)
```

---

as_array	<i>Convert a data cube to an in-memory R array</i>
----------	--

---

### Description

Convert a data cube to an in-memory R array

### Usage

```
as_array(x)
```

### Arguments

x                    data cube

### Value

Four dimensional array with dimensions band, t, y, x

### Note

Depending on the data cube size, this function may require substantial amounts of main memory, i.e. it makes sense for small data cubes only.

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-05"),
              srs="EPSG:32618", nx = 100, ny=100, dt="P1M")
x = as_array(select_bands(raster_cube(L8.col, v), c("B04", "B05")))
dim(x)
dimnames(x)
```

---

`as_json`*Query data cube properties*

---

## Description

`gdalcubes` internally uses a graph to serialize data cubes (including chained operations on cubes). This function derives a JSON representation, which can be used to save data cube objects without pixel data to disk.

## Usage

```
as_json(obj, file = NULL)
```

## Arguments

<code>obj</code>	a data cube proxy object (class <code>cube</code> )
<code>file</code>	optional output file

## Value

If `file` is `NULL`, the function returns a JSON string representing a graph that can be used to recreate the same chain of `gdalcubes` operations even in a different R sessions.

## Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8-L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-04"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
cat(as_json(select_bands(raster_cube(L8.col, v), c("B04", "B05"))))
```

---

bands	<i>Query data cube properties</i>
-------	-----------------------------------

---

**Description**

Query data cube properties

**Usage**

```
bands(obj)
```

**Arguments**

obj                    a data cube proxy object (class cube)

**Value**

A data.frame with rows representing the bands and columns representing properties of a band (name, type, scale, offset, unit)

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
bands(raster_cube(L8.col, v))
```

---

chunk_apply	<i>Apply an R function on chunks of a data cube</i>
-------------	---

---

**Description**

Apply an R function on chunks of a data cube

**Usage**

```
chunk_apply(cube, f)
```

**Arguments**

cube	source data cube
f	R function to apply over all chunks

**Details**

This function internally creates a gdalcubes stream data cube, which streams data of a chunk to a new R process. For reading data, the function typically calls `x <- read_chunk_as_array()` which then results in a 4 dimensional (band, time, y, x) array. Similarly `write_chunk_from_array(x)` will write a result array as a chunk in the resulting data cube. The chunk size of the input cube is important to control how the function will be exposed to the data cube. For example, if you want to apply an R function over complete pixel time series, you must define the chunk size argument in `raster_cube` to make sure that chunk contain the correct parts of the data.

**Value**

a proxy data cube object

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")

L8.cube = raster_cube(L8.col, v)
L8.cube = select_bands(L8.cube, c("B04", "B05"))
f <- function() {
  x <- read_chunk_as_array()
  out <- reduce_time(x, function(x) {
    cor(x[1,], x[2,], use="na.or.complete", method = "kendall")
  })
  write_chunk_from_array(out)
}
L8.cor = chunk_apply(L8.cube, f)
```

---

collection\_formats      *List predefined image collection formats*

---

### Description

gdalcubes comes with some predefined collection formats e.g. to scan Sentinel 2 data. This function lists available formats including brief descriptions.

### Usage

```
collection_formats(print = TRUE)
```

### Arguments

print	logical; should available formats and their descriptions be printed nicely, defaults to TRUE
-------	--

### Details

Image collection formats define how individual files / GDAL datasets relate to an image collection, i.e., which bands they contain, to which image they belong, and how to derive acquisition date/time. They are described as a set of regular expressions in a JSON file and used by gdalcubes to extract this information from the paths and/or filenames.

### Value

data.frame with columns name and description where the former describes the unique identifier that can be used in `create_image_collection` and the latter gives a brief description of the format.

### Examples

```
collection_formats()
```

---

`create_image_collection`  
*Create an image collection from a set of GDAL datasets or files*

---

### Description

This function iterates over files or GDAL dataset identifiers and extracts datetime, image identifiers, and band information according to a given collection format.

**Usage**

```

create_image_collection(
  files,
  format = NULL,
  out_file = tempfile(fileext = ".sqlite"),
  date_time = NULL,
  band_names = NULL,
  use_subdatasets = FALSE,
  unroll_archives = TRUE,
  quiet = FALSE,
  one_band_per_file = NULL
)

```

**Arguments**

files	character vector with paths to image files on disk or any GDAL dataset identifiers (including virtual file systems and higher level drivers or GDAL subdatasets)
format	collection format, can be either a name to use predefined formats (as output from <a href="#">collection_formats</a> ) or a path to a custom JSON format description file
out_file	optional name of the output SQLite database file, defaults to a temporary file
date_time	vector with date/ time for files; can be of class character, Date, or POSIXct (argument is only applicable for image collections without collection format)
band_names	character vector with band names, length must match the number of bands in provided files (argument is only applicable for image collections without collection format)
use_subdatasets	logical; use GDAL subdatasets of provided files (argument is only applicable for image collections without collection format)
unroll_archives	automatically convert .zip, .tar archives and .gz compressed files to GDAL virtual file system dataset identifiers (e.g. by prepending /vsizip/) and add contained files to the list of considered files
quiet	logical; if TRUE, do not print resulting image collection if return value is not assigned to a variable
one_band_per_file	logical; if TRUE, assume that band_names are given for all files (argument is only applicable for image collections without collection format, see Details)

**Details**

An image collection is a simple index (a SQLite database) containing references to existing image files / GDAL dataset identifiers.

Collections can be created in two different ways: First, if a collection format is specified (argument format), date/time, bands, and metadata are automatically extracted from provided files. This is the most general approach but requires a collection format for the specific dataset.



Second, image collections can sometimes be created without collection format by manually specifying date/time of images (argument `date_time`) and names of bands (argument `band_names`). This is possible if either each image file contains *all* bands of the collection or only a single band. In the former case `band_names` simply contains the names of the bands or can be `NULL` to use default names. In the latter case (image files contain a single band only), the lengths of `band_names` and `date_time` must be identical. By default, the function assumes one band per file if `length(band_names) == length(files)`. In the unlikely situation that this is not desired, it can be explicitly set using `one_band_per_file`.

### Value

image collection proxy object, which can be used to create a data cube using `raster_cube`

### Examples

```
# 1. create image collection using a collection format
L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                      ".TIF", recursive = TRUE, full.names = TRUE)
x = create_image_collection(L8_files, "L8_L1TP")
x

# 2. create image collection without format for a single band
L8_files_B4 <- list.files(system.file("L8NY18", package = "gdalcubes"),
                          "_B4.TIF", recursive = TRUE, full.names = TRUE)
d = as.Date(substr(basename(L8_files_B4), 18, 25), "%Y%m%d")
y = create_image_collection(L8_files_B4, date_time = d, band_names = "B4")
y

# 3. create image collection without format for all bands
d = as.Date(substr(basename(L8_files), 18, 25), "%Y%m%d")
fname = basename(tools::file_path_sans_ext(L8_files))
b = substr(fname, 27, nchar(fname))
z = create_image_collection(L8_files, date_time = d, band_names = b)
z
```

---

crop

*Crop data cube extent by space and/or time*

---

### Description

Create a proxy data cube, which crops a data cube by a spatial and/or temporal extent.

### Usage

```
crop(cube, extent = NULL, iextent = NULL, snap = "near")
```

**Arguments**

cube	source data cube
extent	list with numeric items left, right, top, bottom, and character items t0 and t1, or a subset thereof, see examples
iextent	list with length-two integer items named x, y, and t, defining the lower and upper boundaries as integer coordinates, see examples
snap	one of 'near', 'in', or 'out'; ignored if using iextent

**Details**

The new extent can be specified by spatial coordinates and datetime values (using the extent argument), or as zero-based integer indexes (using the iextent argument). In the former case, extent expects a list with numeric items left, right, top, bottom, t0, and t1, or a subset thereof. In the latter case, iextent is expected as a list with length-two integer vectors x, y, and t as items, defining the lower and upper cell indexes per dimension.

Notice that it is possible to crop only selected boundaries (e.g., only the right boundary) as missing boundaries in the extent or NA / NULL values in the iextent arguments are considered as "no change". It is, however, not possible to mix arguments extent and iextent.

If extent is given, the snap argument can be used to define what happens if the new boundary falls within a data cube cell.

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}
L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                        bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P3M", aggregation = "median")
L8.cube = raster_cube(L8.col, v, mask=image_mask("BQA", bits=4, values=16))
L8.rgb = select_bands(L8.cube, c("B02", "B03", "B04"))

# crop by integer indexes
L8.cropped = crop(L8.rgb, iextent = list(x=c(0,400), y=c(0,400), t=c(1,1)))

# crop by spatiotemporal coordinates
L8.cropped = crop(L8.rgb, extent = list(left=388941.2, right=766552.4,
                                       bottom=4345299, top=4744931, t0="2018-01", t1="2018-06"), snap = "in")
L8.cropped
```

```
L8.cropped = crop(L8.rgb, extent = list(left=388941.2, right=766552.4,
    bottom=4345299, top=4744931, t0="2018-01", t1="2018-06"), snap = "near")
L8.cropped

plot(L8.cropped, rgb = 3:1, zlim=c(5000,10000))
```

---

cube\_view

*Create or update a spatiotemporal data cube view*


---

### Description

Data cube views define the shape of a cube, i.e., the spatiotemporal extent, resolution, and spatial reference system (srs). They are used to access image collections as on-demand data cubes. The data cube will filter images based on the view's extent, read image data at the defined resolution, and warp / reproject images to the target srs automatically.

### Usage

```
cube_view(
  view,
  extent,
  srs,
  nx,
  ny,
  nt,
  dx,
  dy,
  dt,
  aggregation,
  resampling,
  keep.asp = TRUE
)
```

### Arguments

view	if provided, update this cube_view object instead of creating a new data cube view where fields that are already set will be overwritten
extent	spatiotemporal extent as a list e.g. from <a href="#">extent</a> or an image collection object, see Details
srs	target spatial reference system as a string; can be a proj4 definition, WKT, or in the form "EPSG:XXXX"
nx	number of pixels in x-direction (longitude / easting)
ny	number of pixels in y-direction (latitude / northing)

nt	number of pixels in t-direction
dx	size of pixels in x-direction (longitude / easting)
dy	size of pixels in y-direction (latitude / northing)
dt	size of pixels in time-direction, expressed as ISO8601 period string (only 1 number and unit is allowed) such as "P16D"
aggregation	aggregation method as string, defining how to deal with pixels containing data from multiple images, can be "min", "max", "mean", "median", or "first"
resampling	resampling method used in gdalwarp when images are read, can be "near", "bilinear", "bicubic" or others as supported by gdalwarp (see <a href="https://gdal.org/programs/gdalwarp.html">https://gdal.org/programs/gdalwarp.html</a> )
keep.asp	if TRUE, derive ny or dy automatically from nx or dx (or vice versa) based on the aspect ratio of the spatial extent

### Details

The extent argument expects a simple list with elements left, right, bottom, top, t0 (start date/time), t1 (end date/time) or an image collection object. In the latter case, the `extent` function is automatically called on the image collection object to get the full spatiotemporal extent of the collection. In the former case, datetimes are expressed as ISO8601 datetime strings.

The function can be used in two different ways. First, it can create data cube views from scratch by defining the extent, the spatial reference system, and for each dimension either the cell size (dx, dy, dt) or the total number of cells (nx, ny, nt). Second, the function can update an existing data cube view by overwriting specific fields. In this case, the extent or some elements of the extent may be missing.

In some cases, the extent of the view is automatically extended if the provided resolution would end within a pixel. For example, if the spatial extent covers an area of 1km x 1km and dx = dy = 300m, the extent would be enlarged to 1.2 km x 1.2km. The alignment will be reported to the user in a diagnostic message.

### Value

A list with data cube view properties

### Examples

```
L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                      ".TIF", recursive = TRUE, full.names = TRUE)
L8.col = create_image_collection(L8_files, "L8_L1TP")

# 1. Create a new data cube view specification
v = cube_view(extent=extent(L8.col,"EPSG:4326"), srs="EPSG:4326", dt="P1M",
              nx=1000, ny=500, aggregation = "mean", resampling="bilinear")
v

# 2. overwrite parts of an existing data cube view
vnew = cube_view(v, dt="P1M")
```

---

dim.cube	<i>Query data cube properties</i>
----------	-----------------------------------

---

## Description

Query data cube properties

## Usage

```
## S3 method for class 'cube'  
dim(x)
```

## Arguments

x a data cube proxy object (class cube)

## Value

size of a data cube (number of cells) as integer vector in the order t, y, x

## See Also

[size](#)

## Examples

```
# create image collection from example Landsat data only  
# if not already done in other examples  
if (!file.exists(file.path(tempdir(), "L8.db"))) {  
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),  
                        ".TIF", recursive = TRUE, full.names = TRUE)  
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)  
}  
  
L8.col = image_collection(file.path(tempdir(), "L8.db"))  
v = cube_view(extent=list(left=388941.2, right=766552.4,  
                        bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),  
             srs="EPSG:32618", nx = 497, ny=526, dt="P1M")  
dim(raster_cube(L8.col, v))
```

---

dimensions	<i>Query data cube properties</i>
------------	-----------------------------------

---

### Description

Query data cube properties

### Usage

```
dimensions(obj)
```

### Arguments

obj            a data cube proxy object (class cube)

### Details

Elements of the returned list represent individual dimensions with properties such as dimension boundaries, names, and chunk size stored as inner lists

### Value

Dimension information as a list

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                        bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
dimensions(raster_cube(L8.col, v))
```

---

dimension\_bounds      *Query coordinate bounds for all dimensions of a data cube*

---

**Description**

Dimension values give the coordinates bounds the spatial and temporal axes of a data cube.

**Usage**

```
dimension_bounds(obj, datetime_unit = NULL)
```

**Arguments**

`obj`                    a data cube proxy (class cube)  
`datetime_unit`        unit used to format values in the datetime dimension, one of "Y", "m", "d", "H", "M", "S", defaults to the unit of the cube.

**Value**

list with elements t,y,x, each a list with two elements, start and end

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
dimension_bounds(raster_cube(L8.col, v))
```

---

dimension\_values      *Query coordinate values for all dimensions of a data cube*

---

**Description**

Dimension values give the coordinates along the spatial and temporal axes of a data cube.

**Usage**

```
dimension_values(obj, datetime_unit = NULL)
```

**Arguments**

`obj` a data cube proxy (class `cube`), or a data cube view object

`datetime_unit` unit used to format values in the datetime dimension, one of "Y", "m", "d", "H", "M", "S", defaults to the unit of the cube.

**Value**

list with elements `t,y,x`

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
dimension_values(raster_cube(L8.col, v))
```

---

extent

*Derive the spatiotemporal extent of an image collection*

---

**Description**

Derive the spatiotemporal extent of an image collection

**Usage**

```
extent(x, srs = "EPSG:4326")
```

**Arguments**

`x` image collection proxy object

`srs` target spatial reference system

**Value**

a list with elements `left`, `right`, `bottom`, `top`, `t0` (start date/time), and `t1` (end date/time)



## Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
extent(L8.col, "EPSG:32618")
cube_view(extent=extent(L8.col, "EPSG:32618"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
```

---

extract\_geom

*Extract values from a data cube by spatial or spatiotemporal features*

---

## Description

Extract pixel values of a data cube from a set of spatial or spatiotemporal features. Applications include the extraction of full time series at irregular points, extraction from spatiotemporal points, extraction of pixel values in polygons, and computing summary statistics over polygons.

## Usage

```
extract_geom(
  cube,
  sf,
  datetime = NULL,
  time_column = NULL,
  FUN = NULL,
  merge = FALSE,
  drop_geom = FALSE,
  ...,
  reduce_time = FALSE
)
```

## Arguments

cube	source data cube to extract values from
sf	object of class sf, see <a href="#">sf package</a>
datetime	Date, POSIXt, or character vector containing per feature time information; length must be identical to the number of features in sf
time_column	name of the column in sf containing per feature time information
FUN	optional function to compute per feature summary statistics

merge	logical; return a combined data.frame with data cube values and labels, defaults to FALSE
drop_geom	logical; remove geometries from output, only used if merge is TRUE, defaults to FALSE
...	additional arguments passed to FUN
reduce_time	logical; if TRUE, time is ignored when FUN is applied

## Details

The geometry in `sf` can be of any simple feature type supported by GDAL, including POINTS, LINES, POLYGONS, MULTI\*, and more. If no time information is provided in one of the arguments `datetime` or `time_column`, the full time series of pixels with regard to the features are returned.

Notice that feature identifiers in the FID column typically correspond to the row names / numbers of the provided `sf` object. This can be used to combine the output with the original geometries, e.g., using `merge()`. with `gdalcubes > 0.6.4`, this can be done automatically by setting `merge=TRUE`. In this case, the FID column is dropped from the result.

Pixels with missing values are automatically dropped from the result. It is hence not guaranteed that the result will contain rows for all input features.

Features are automatically reprojected if the coordinate reference system differs from the data cube.

Extracted values can be aggregated by features by providing a summary function. If `reduce_time` is FALSE (the default), the values are grouped by feature and time, i.e., the result will contain unique combinations of FID and time. To ignore time and produce a single value per feature, `reduce_time` can be set to TRUE.

## Value

A data.frame with columns FID, time, and data cube bands / variables, see Details

## Examples

```
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}
L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(srs="EPSG:32618", dy=1000, dx=1000, dt="P1M",
  aggregation = "median", resampling = "bilinear",
  extent=list(left=388941.2, right=766552.4,
    bottom=4345299, top=4744931,
    t0="2018-01-01", t1="2018-04-30"))
L8.cube = raster_cube(L8.col, v)
L8.cube = select_bands(L8.cube, c("B04", "B05"))
L8.ndvi = apply_pixel(L8.cube, "(B05-B04)/(B05+B04)", "NDVI")
L8.ndvi
```

```

if (gdalcubes_gdal_has_geos()) {
  if (requireNamespace("sf", quietly = TRUE)) {

    # create 50 random point locations
    x = runif(50, v$space$left, v$space$right)
    y = runif(50, v$space$bottom, v$space$top)
    t = sample(seq(as.Date("2018-01-01"),as.Date("2018-04-30")), by = 1),50, replace = TRUE)
    df = sf::st_as_sf(data.frame(x = x, y = y), coords = c("x", "y"), crs = v$space$srs)

    # 1. spatiotemporal points
    extract_geom(L8.ndvi, df, datetime = t)

    # 2. time series at spatial points
    extract_geom(L8.ndvi, df)

    # 3. summary statistics over polygons
    x = sf::st_read(system.file("nycd.gpkg", package = "gdalcubes"))
    zstats = extract_geom(L8.ndvi,x, FUN=median, reduce_time = TRUE, merge = TRUE)
    zstats
    plot(zstats["NDVI"])

  }
}

```

---

fill\_time

---

*Fill NA data cube pixels by simple time series interpolation*


---

### Description

Create a proxy data cube, which fills NA pixels of a data cube by nearest neighbor or linear time series interpolation.

### Usage

```
fill_time(cube, method = "near")
```

### Arguments

cube	source data cube
method	interpolation method, can be "near" (nearest neighbor), "linear" (linear interpolation), "lof" (last observation carried forward), or "nocb" (next observation carried backward)

### Details

Please notice that completely empty (NA) time series will not be filled, i.e. the result cube might still contain NA values.

**Value**

a proxy data cube object

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}
L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P3M", aggregation = "median")
L8.cube = raster_cube(L8.col, v, mask=image_mask("BQA", bits=4, values=16))
L8.rgb = select_bands(L8.cube, c("B02", "B03", "B04"))
L8.filled = fill_time(L8.rgb, "linear")
L8.filled

plot(L8.filled, rgb=3:1, zlim=c(5000,12000))
```

---

filter\_geom

*Filter data cube pixels by a polygon*

---

**Description**

Create a proxy data cube, which filters pixels by a spatial (multi)polygon For all pixels whose center is within the polygon, the original

**Usage**

```
filter_geom(cube, geom, srs = NULL)
```

**Arguments**

cube	source data cube
geom	either a WKT string, or an sfc or sfg object (sf package)
srs	string identifier of the polygon's coordinate reference system understandable for GDAL

**Details**

The resulting data cube will not be cropped but pixels outside of the polygon will be set to NAN.

If geom is provided as an sfc object with length > 1, geometries will be combined with `sf::st_combine()` before.

The geometry is automatically transformed to the data cube's spatial reference system if needed.

**Value**

a proxy data cube object

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-06"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
L8.cube = raster_cube(L8.col, v)
L8.cube = select_bands(L8.cube, c("B04", "B05"))
L8.ndvi = apply_pixel(L8.cube, "(B05-B04)/(B05+B04)", "NDVI")
WKT = gsub(pattern='\\n',replacement="",x =
  "Polygon ((-74.3541 40.9254,
    -73.9813 41.2467,
    -73.9997 41.4400,
    -74.5362 41.1795,
    -74.6286 40.9137,
    -74.3541 40.9254)))")
L8.ndvi.filtered = filter_geom(L8.ndvi, WKT, "EPSG:4326")
L8.ndvi.filtered

plot(L8.ndvi.filtered)
```

---

 filter\_pixel

*Filter data cube pixels by a user-defined predicate on band values*


---

### Description

Create a proxy data cube, which evaluates a predicate over all pixels of a data cube. For all pixels that fulfill the predicate, the original band values are returned. Other pixels are simply filled with NaNs. The predicate may access band values by name.

### Usage

```
filter_pixel(cube, pred)
```

### Arguments

cube	source data cube
pred	predicate to be evaluated over all pixels

### Details

gdalcubes uses and extends the [tinyexpr library](#) to evaluate expressions in C / C++, you can look at the [library documentation](#) to see what kind of expressions you can execute. Pixel band values can be accessed by name.

### Value

a proxy data cube object

### Note

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-01", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
L8.cube = raster_cube(L8.col, v)
L8.cube = select_bands(L8.cube, c("B04", "B05"))
```

```
L8.ndvi = apply_pixel(L8.cube, "(B05-B04)/(B05+B04)", "NDVI")
L8.ndvi.filtered = filter_pixel(L8.ndvi, "NDVI > 0.5")
L8.ndvi.filtered

plot(L8.ndvi.filtered)
```

---

gdalcubes

*gdalcubes: Earth Observation Data Cubes from Satellite Image Collections*

---

## Description

Processing collections of Earth observation images as on-demand multispectral, multitemporal raster data cubes. Users define cubes by spatiotemporal extent, resolution, and spatial reference system and let 'gdalcubes' automatically apply cropping, reprojection, and resampling using the 'Geospatial Data Abstraction Library' ('GDAL'). Implemented functions on data cubes include reduction over space and time, applying arithmetic expressions on pixel band values, moving window aggregates over time, filtering by space, time, bands, and predicates on pixel values, exporting data cubes as 'netCDF' or 'GeoTIFF' files, plotting, and extraction from spatial and or spatiotemporal features. All computational parts are implemented in C++, linking to the 'GDAL', 'netCDF', 'CURL', and 'SQLite' libraries. See Appel and Pebesma (2019) <doi:10.3390/data4030092> for further details.

---

gdalcubes\_gdalformats *Get available GDAL drivers*

---

## Description

Get available GDAL drivers

## Usage

```
gdalcubes_gdalformats()
```

## Examples

```
gdalcubes_gdalformats()
```

gdalcubes\_gdalversion *Get the GDAL version used by gdalcubes*

---

**Description**

Get the GDAL version used by gdalcubes

**Usage**

```
gdalcubes_gdalversion()
```

**Examples**

```
gdalcubes_gdalversion()
```

---

gdalcubes\_gdal\_has\_geos  
*Check if GDAL was built with GEOS*

---

**Description**

Check if GDAL was built with GEOS

**Usage**

```
gdalcubes_gdal_has_geos()
```

**Examples**

```
gdalcubes_gdal_has_geos()
```

---

gdalcubes\_options *Set or read global options of the gdalcubes package*

---

**Description**

Set global package options to change the default behavior of gdalcubes. These include how many parallel processes are used to process data cubes, how created netCDF files are compressed, and whether or not debug messages should be printed.



**Usage**

```

gdalcubes_options(
  ...,
  parallel,
  ncdf_compression_level,
  debug,
  cache,
  ncdf_write_bounds,
  use_overview_images,
  show_progress,
  default_chunksize,
  streaming_dir,
  log_file,
  threads
)

```

**Arguments**

...	not used
parallel	number of parallel workers used to process data cubes or TRUE to use the number of available cores automatically
ncdf_compression_level	integer; compression level for created netCDF files, 0=no compression, 1=fast compression, 9=small compression
debug	logical; print debug messages
cache	logical; TRUE if temporary data cubes should be cached to support fast reprocessing of the same cubes
ncdf_write_bounds	logical; write dimension bounds as additional variables in netCDF files
use_overview_images	logical; if FALSE, all images are read on original resolution and existing overviews will be ignored
show_progress	logical; if TRUE, a progress bar will be shown for actual computations
default_chunksize	length-three vector with chunk size in t, y, x directions or a function taking a data cube size and returning a suggested chunk size
streaming_dir	directory where temporary binary files for process streaming will be written to
log_file	character, if empty string or NULL, diagnostic messages will be printed to the console, otherwise to the provided file
threads	number of threads used to process data cubes (deprecated)

**Details**

Data cubes can be processed in parallel where the number of chunks in a cube is distributed among parallel worker processes. The actual number of used workers can be lower if a data cube as less

chunks. If `parallel` is `TRUE`, the number of available cores is used. Setting `parallel = FALSE` can be used to disable parallel processing. Notice that since version 0.6.0, separate processes are being used instead of parallel threads to avoid possible R session crashes due to some multithreading issues.

Caching has no effect on disk or memory consumption, it simply tries to reuse existing temporary files where possible. For example, changing only parameters to `plot` will void reprocessing the same data cube if `cache` is `TRUE`.

The streaming directory can be used to control the performance of user-defined functions, if disk IO is a bottleneck. Ideally, this can be set to a directory on a shared memory device.

Passing no arguments will return the current options as a list.

## Examples

```
gdalcubes_options(parallel=4) # set the number
gdalcubes_options() # print current options
gdalcubes_options(parallel=FALSE) # reset
```

---

gdalcubes\_selection    *Subsetting data cubes*

---

## Description

Subset data cube dimensions and bands / variables.

## Usage

```
## S3 method for class 'cube'
x$name

## S3 method for class 'cube'
x[ib = TRUE, it = TRUE, iy = TRUE, ix = TRUE, ...]
```

## Arguments

<code>x</code>	source data cube
<code>name</code>	character; name of selected band
<code>ib</code>	first selector (optional), object of type character, list, Date, POSIXt, numeric, <a href="#">st_bbox</a> , or <a href="#">st_sfc</a> , see Details and examples
<code>it</code>	second selector (optional), see <code>ib</code>
<code>iy</code>	third selector (optional), see <code>ib</code>
<code>ix</code>	fourth selector (optional), see <code>ib</code>
<code>...</code>	further arguments, not used

## Details

The `[]` operator allows for flexible subsetting of data cubes by date, datetime, bounding box, spatial points, and band names. Depending on the arguments, it supports slicing (selecting one element of a dimension), cropping (selecting a subinterval of a dimension) and combinations thereof (e.g., selecting a spatial window and a temporal slice). Dimension subsets can be specified by integer indexes or coordinates / datetime values. Arguments are matched by type and order. For example, if the first argument is a length-two vector of type `Date`, the function will understand to subset the time dimension. Otherwise, arguments are treated in the order band, time, y, x.

## Note

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

## Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}
L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P3M", aggregation = "median")
L8.cube = raster_cube(L8.col, v, mask=image_mask("BQA", bits=4, values=16))
L8.red = L8.cube$B04

plot(L8.red)

v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-01-01", t1="2018-12-31"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1D", aggregation = "median")
L8.cube = raster_cube(L8.col, v, mask=image_mask("BQA", bits=4, values=16))

L8.cube[c("B05", "B04")] # select bands
L8.cube[as.Date(c("2018-01-10", "2018-01-20"))] # crop by time
L8.cube[as.Date("2018-01-10")] # slice by time
L8.cube["B05", "2018-01-10"] # select bands and slice by time
L8.cube["B05", c("2018-01-10", "2018-01-17")] # select bands and crop by time
L8.cube[, c("2018-01-10", "2018-01-17")] # crop by time

# crop by space (coordinates and integer indexes respectively)
L8.cube[list(left=388941.2 + 1e5, right=766552.4 - 1e5, bottom=4345299 + 1e5, top=4744931 - 1e5)]
L8.cube[, , c(1,100), c(1,100)]

L8.cube[, c(1,2), , ] # crop by time (integer indexes)
```

```
# subset by spatial point or bounding box
if (requireNamespace("sf", quietly = TRUE)) {
  s = sf::st_sfc(sf::st_point(c(500000, 4500000)), crs = "EPSG:32618")
  L8.cube[s]

  bbox = sf::st_bbox(c(xmin = 388941.2 + 1e5, xmax = 766552.4 - 1e5,
                        ymax = 4744931 - 1e5, ymin = 4345299 + 1e5), crs = sf::st_crs(32618))
  L8.cube[bbox]
}
```

---

gdalcubes\_set\_gdal\_config

*Set GDAL config options*

---

## Description

Set GDAL config options

## Usage

```
gdalcubes_set_gdal_config(key, value)
```

## Arguments

key	name of a GDAL config option to be set
value	value

## Details

Details and a list of possible options can be found at <https://gdal.org/user/configoptions.html>.

## Examples

```
gdalcubes_set_gdal_config("GDAL_NUM_THREADS", "ALL_CPUS")
```

---

image_collection	<i>Load an existing image collection from a file</i>
------------------	--

---

### Description

This function will load an image collection from an SQLite file. Image collection files index and reference existing imagery. To create a collection from files on disk, use [create\\_image\\_collection](#).

### Usage

```
image_collection(path)
```

### Arguments

path	path to an existing image collection file
------	---

### Value

an image collection proxy object, which can be used to create a data cube using [raster\\_cube](#)

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
L8.col
```

---

image_mask	<i>Create a mask for images in a raster data cube</i>
------------	---

---

### Description

Create an image mask based on a band and provided values to filter pixels of images read by [raster\\_cube](#)

## Usage

```
image_mask(  
  band,  
  min = NULL,  
  max = NULL,  
  values = NULL,  
  bits = NULL,  
  invert = FALSE  
)
```

## Arguments

band	name of the mask band
min	minimum value, values between min and max will be masked
max	maximum value, values between min and max will be masked
values	numeric vector; specific values that will be masked.
bits	for bitmasks, extract the given bits (integer vector) with a bitwise AND before filtering the mask values, bit indexes are zero-based
invert	logical; invert mask

## Details

Values of the selected mask band can be based on a range (by passing min and max) or on a set of values (by passing values). By default pixels with mask values contained in the range or in the values are masked out, i.e. set to NA. Setting invert = TRUE will invert the masking behavior. Passing values will override min and max.

## Note

Notice that masks are applied per image while reading images as a raster cube. They can be useful to eliminate e.g. cloudy pixels before applying the temporal aggregation to merge multiple values for the same data cube pixel.

## Examples

```
image_mask("SCL", values = c(3,8,9)) # Sentinel 2 L2A: mask cloud and cloud shadows  
image_mask("BQA", bits=4, values=16) # Landsat 8: mask clouds  
image_mask("B10", min = 8000, max=65000)
```

---

join_bands	<i>Join bands of two identically shaped data cubes</i>
------------	--

---

### Description

Create a proxy data cube, which joins the bands of two identically shaped data cubes. The resulting cube will have bands from both input cubes.

### Usage

```
join_bands(cube_list, cube_names = NULL)
```

### Arguments

cube_list	a list with two or more source data cubes
cube_names	list or character vector with optional name prefixes for bands in the output data cube (see Details)

### Details

The number of provided cube\_names must match the number of provided input cubes. If no cube\_names are provided, bands of the output cube will adopt original names from the input cubes (without any prefix). If any two of the input bands have identical names, prefixes default prefixes ("X1", "X2", ...) will be used.

### Value

proxy data cube object

### Note

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-01", t1="2018-05"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
```

```

L8.cube = raster_cube(L8.col, v)
L8.cube.b04 = select_bands(raster_cube(L8.col, v), c("B04"))
L8.cube.b05 = select_bands(raster_cube(L8.col, v), c("B05"))
join_bands(list(L8.cube.b04,L8.cube.b05))

plot(join_bands(list(L8.cube.b04,L8.cube.b05)))

```

---

json\_cube

*Read a data cube from a json description file*

---

## Description

Read a data cube from a json description file

## Usage

```
json_cube(json, path = NULL)
```

## Arguments

json	length-one character vector with a valid json data cube description
path	source data cube proxy object

## Details

Data cubes can be stored as JSON description files. These files do not store any data but the recipe how a data cube is constructed, i.e., the chain (or graph) of processes involved.

Since data cube objects (as returned from [raster\\_cube](#)) cannot be saved with normal R methods, the combination of [as\\_json](#) and [json\\_cube](#) provides a cheap way to save virtual data cube objects across several R sessions, as in the examples.

## Value

data cube proxy object

## Examples

```

# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),

```



```

        srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
cube = raster_cube(L8.col, v)

# save
fname = tempfile()
as_json(cube, fname)

# load
json_cube(path = fname)

```

---

memsize

*Query data cube properties*


---

## Description

Query data cube properties

## Usage

```
memsize(obj, unit = "MiB")
```

## Arguments

obj	a data cube proxy object (class cube)
unit	Unit of data size, can be "B", "KB", "KiB", "MB", "MiB", "GB", "GiB", "TB", "TiB", "PB", "PiB"

## Value

Total data size of data cube values expressed in the given unit

## Examples

```

# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
memsize(raster_cube(L8.col, v))

```

---

names.cube                      *Query data cube properties*

---

### Description

Query data cube properties

### Usage

```
## S3 method for class 'cube'
names(x)
```

### Arguments

x                      a data cube proxy object (class cube)

### Value

Band names as character vector

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
names(raster_cube(L8.col, v))
```

---

nbands                              *Query data cube properties*

---

### Description

Query data cube properties

### Usage

```
nbands(obj)
```

**Arguments**

obj                    a data cube proxy object (class cube)

**Value**

Number of bands

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
nbands(raster_cube(L8.col, v))
```

---

ncdf\_cube

*Read a data cube from an existing netCDF file*


---

**Description**

Create a proxy data cube from a netCDF file that has been created using [write\\_ncdf](#). This function does not read cubes from arbitrary netCDF files and can be used e.g., to load intermediate results and/or plotting existing netCDF cubes on disk without doing the data cube creation from image collections.

**Usage**

```
ncdf_cube(path, chunking = NULL, auto_unpack = TRUE)
```

**Arguments**

path                    path to an existing netCDF file

chunking                custom chunk sizes to read from the netCDF file; defaults to using chunk sizes from the netCDF file

auto\_unpack            logical; automatically apply offset and scale when reading data values

**Value**

a proxy data cube object

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")

ncfile = write_ncdf(select_bands(raster_cube(L8.col, v), c("B02", "B03", "B04")))
ncdf_cube(ncfile)
```

---

nt

*Query data cube properties*

---

**Description**

Query data cube properties

**Usage**

```
nt(obj)
```

**Arguments**

obj                    a data cube proxy object (class cube)

**Value**

Number of pixels in the time dimension

## Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
nt(raster_cube(L8.col, v))
```

---

nx *Query data cube properties*

---

## Description

Query data cube properties

## Usage

```
nx(obj)
```

## Arguments

obj                    a data cube proxy object (class cube)

## Value

Number of pixels in the x dimension

## Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
nx(raster_cube(L8.col, v))
```

---

ny *Query data cube properties*

---

### Description

Query data cube properties

### Usage

ny(obj)

### Arguments

obj a data cube proxy object (class cube)

### Value

Number of pixels in the y dimension

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
ny(raster_cube(L8.col, v))
```

---

pack\_minmax *Helper function to define packed data exports by min / max values*

---

### Description

This function can be used to define packed exports in [write\\_ncdf](#) and [write\\_tif](#). It will generate scale and offset values with maximum precision (unless simplify=TRUE).

### Usage

```
pack_minmax(type = "int16", min, max, simplify = FALSE)
```

**Arguments**

type	target data type of packed values (one of "uint8", "uint16", "uint32", "int16", or "int32")
min	numeric; minimum value(s) of original values, will be packed to the 2nd lowest value of the target data type
max	numeric; maximum value(s) in original scale, will be packed to the highest value of the target data type
simplify	logical; round resulting scale and offset to power of 10 values

**Details**

Nodata values will be mapped to the lowest value of the target data type.

Arguments min and max must have length 1 or length equal to the number of bands of the data cube to be exported. In the former case, the same values are used for all bands of the exported target cube, whereas the latter case allows to use different ranges for different bands.

**Note**

Using simplify=TRUE will round scale values to the next smaller power of 10.

**Examples**

```
ndvi_packing = pack_minmax(type="int16", min=-1, max=1)
ndvi_packing
```

---

plot.cube

*Plot a gdalcubes data cube*


---

**Description**

Plot a gdalcubes data cube

**Usage**

```
## S3 method for class 'cube'
plot(
  x,
  y,
  ...,
  nbreaks = 11,
  breaks = NULL,
  col = grey(1:(nbreaks - 1)/nbreaks),
  key.pos = NULL,
  bands = NULL,
  t = NULL,
```

```

    rgb = NULL,
    zlim = NULL,
    gamma = 1,
    periods.in.title = TRUE,
    join.timeseries = FALSE,
    axes = TRUE,
    ncol = NULL,
    nrow = NULL,
    downsample = TRUE,
    na.color = "#AAAAAA"
)

```

### Arguments

x	a data cube proxy object (class cube)
y	<u>not used</u>
...	further arguments passed to <code>image.default</code>
nbreaks	number of breaks, should be one more than the number of colors given
breaks	actual breaks used to assign colors to values; if missing, the function subsamples values and uses equally sized intervals between min and max or <code>zlim[0]</code> and <code>zlim[1]</code> if defined
col	color definition, can be a character vector with <code>nbreaks - 1</code> elements or a function such as <code>heat.colors</code>
key.pos	position for the legend, 1 (bottom), 2 (left), 3 (top), or 4 (right). If NULL (the default), do not plot a legend.
bands	integer vector with band numbers to plot (this must be band numbers, not band names)
t	integer vector with time indexes to plot (this must be time indexes, not date / time)
rgb	bands used to assign RGB color channels, vector of length 3 (this must be band numbers, not band names)
zlim	vector of length 2, defining the minimum and maximum values to either derive breaks, or define black and white values in RGB plots
gamma	gamma correction value, used for RGB plots only
periods.in.title	logical value, if TRUE, the title of plots includes the datetime period length as ISO 8601 string
join.timeseries	logical, for pure time-series plots, shall time series of multiple bands be plotted in a single plot (with different colors)?
axes	logical, if TRUE, plots include axes
ncol	number of columns for arranging plots with <code>layout()</code> , see Details
nrow	number of rows for arranging plots with <code>layout()</code> , see Details



downsample	length-one integer or logical value used to select only every i-th pixel (in space only) for faster plots; by default (TRUE), downsampling will be determined automatically based on the resolution of the graphics device; set to FALSE to avoid downsampling.
na.color	color used to plot NA pixels

### Details

The style of the plot depends on provided parameters and on the shape of the cube, i.e., whether it is a pure time series and whether it contains multiple bands or not. Multi-band, multi-temporal images will be arranged with `layout()` such that bands are represented by columns and time is represented by rows. Time series plots can be combined to a single plot by setting `join.timeseries = TRUE`. The layout can be controlled with `ncol` and `nrow`, which define the number of rows and columns in the plot layout. Typically, only one of `ncol` and `nrow` is provided. For multi-band, multi-temporal plots, the actual number of rows or columns can be less if the input cube has less bands or time slices.

The `downsample` argument is used to speed-up plotting if the cube has much more pixels than the graphics device. If set to a scalar integer value  $> 1$ , the value is used to skip pixels in the spatial dimensions. For example, setting `downsample = 4` means that every fourth pixel is used in the spatial dimensions. If TRUE (the default) `downsample` is derived automatically based on the sizes of the cube and the graphics device. If 1 or FALSE, no additional downsampling is performed. Notice that downsampling is only used for plotting. The size of the data cube (and hence the computation time to process the data cube) is not modified.

### Note

If caching is enabled for the package (see [gdalcubes\\_options](#)), repeated calls of `plot` for the same data cube will not reevaluate the cube. Instead, the temporary result file will be reused, if possible.

Some parts of the function have been copied from the stars package (c) Edzer Pebesma

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")

plot(select_bands(raster_cube(L8.col, v), c("B02", "B03", "B04")), rgb=3:1)

L8.cube = select_bands(raster_cube(L8.col, v), c("B04", "B05"))
L8.ndvi = apply_pixel(L8.cube, "(B05-B04)/(B05+B04)", "NDVI")
plot(reduce_time(L8.ndvi, "median(NDVI)"), key.pos=1, zlim=c(0,1))
```

---

predict.cube	<i>Model prediction</i>
--------------	-------------------------

---

### Description

Apply a trained model on all pixels of a data cube.

### Usage

```
## S3 method for class 'cube'
predict(object, model, ..., output_names = c("pred"), keep_bands = FALSE)
```

### Arguments

object	a data cube proxy object (class cube)
model	model used for prediction (e.g. from caret or tidymodels)
...	further arguments passed to the model-specific predict method
output_names	optional character vector for output variable(s)
keep_bands	logical; keep bands of input data cube, defaults to FALSE, i.e. original bands will be dropped

### Details

The model-specific predict method will be automatically chosen based on the class of the provided model. It aims at supporting models from the packages tidymodels, caret, and simple models as from lm or glm.

For multiple output variables or output in form of lists or data.frames, output\_names must be provided and match names of the columns / items of the result object returned from the underlying predict method. For example, predictions using tidymodels return a tibble (data.frame) with columns like .pred\_class (classification case). This must be explicitly provided as output\_names. Similarly, predict.lm and the like return lists if the standard error is requested by the user and output\_names hence should be set to c("fit", "se.fit").

For more complex cases or when predict expects something else than a data.frame, this function may not work at all.

### Note

This function returns a proxy object, i.e., it will not immediately start any computations.

**Examples**

```

# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P3M")

L8.col = image_collection(file.path(tempdir(), "L8.db"))

x = sf::st_read(system.file("ny_samples.gpkg", package = "gdalcubes"))

raster_cube(L8.col, v) |>
  select_bands(c("B02", "B03", "B04", "B05")) |>
  extract_geom(x) -> train

x$FID = rownames(x)
train = merge(train, x, by = "FID")
train$iswater = as.factor(train$class == "water")

log_model <- glm(iswater ~ B02 + B03 + B04 + B05, data = train, family = "binomial")

raster_cube(L8.col, v) |>
  select_bands(c("B02", "B03", "B04", "B05")) |>
  predict(model=log_model, type="response") |>
  plot(key.pos=1)

```

---

print.cube

*Print data cube information*


---

**Description**

Prints information about the dimensions and bands of a data cube.

**Usage**

```

## S3 method for class 'cube'
print(x, ...)

```

**Arguments**

x	Object of class "cube"
...	Further arguments passed to the generic print function

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
print(raster_cube(L8.col, v))
```

---

```
print.cube_view      Print data cube view information
```

---

**Description**

Prints information about a data cube view, including its dimensions, spatial reference, aggregation method, and resampling method.

**Usage**

```
## S3 method for class 'cube_view'
print(x, ...)
```

**Arguments**

```
x          Object of class "cube_view"
...        Further arguments passed to the generic print function
```

**Examples**

```
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
print(v)
```

---

```
print.image_collection
```

*Print image collection information*

---

**Description**

Prints information about images in an image collection.

**Usage**

```
## S3 method for class 'image_collection'  
print(x, ..., n = 6)
```

**Arguments**

x	Object of class "image_collection"
...	Further arguments passed to the generic print function
n	Number of images for which details are printed

**Examples**

```
# create image collection from example Landsat data only  
# if not already done in other examples  
if (!file.exists(file.path(tempdir(), "L8.db"))) {  
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),  
                        ".TIF", recursive = TRUE, full.names = TRUE)  
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)  
}  
  
L8.col = image_collection(file.path(tempdir(), "L8.db"))  
print(L8.col)
```

---

```
proj4
```

*Query data cube properties*

---

**Description**

Query data cube properties

**Usage**

```
proj4(obj)
```

**Arguments**

obj	a data cube proxy object (class cube)
-----	---------------------------------------

**Value**

The spatial reference system expressed as proj4 string

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
proj4(raster_cube(L8.col, v))
```

---

raster\_cube

*Create a data cube from an image collection*


---

**Description**

Create a proxy data cube, which loads data from a given image collection according to a data cube view

**Usage**

```
raster_cube(
  image_collection,
  view,
  mask = NULL,
  chunking = .pkgenv$default_chunksize,
  incomplete_ok = TRUE
)
```

**Arguments**

image_collection	Source image collection as from <code>image_collection</code> or <code>create_image_collection</code>
view	A data cube view defining the shape (spatiotemporal extent, resolution, and spatial reference), if missing, a default overview is used
mask	mask pixels of images based on band values, see <a href="#">image_mask</a>
chunking	length-3 vector or a function returning a vector of length 3, defining the size of data cube chunks in the order time, y, x.
incomplete_ok	logical, if TRUE (the default), chunks will ignore IO failures and simply use as much images as possible, otherwise the result will contain empty chunks if IO errors or similar occur.

**Details**

The following steps will be performed when the data cube is requested to read data of a chunk:

1. Find images from the input collection that intersect with the spatiotemporal extent of the chunk
2. For all resulting images, apply gdalwarp to reproject, resize, and resample to an in-memory GDAL dataset
3. Read the resulting data to the chunk buffer and optionally apply a mask on the result
4. Update pixel-wise aggregator (as defined in the data cube view) to combine values of multiple images within the same data cube pixels

If chunking is provided as a function, it must accept exactly three arguments for the total size of the cube in t, y, and x axes (in this order).

**Value**

A proxy data cube object

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                        bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
             srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
raster_cube(L8.col, v)

# using a mask on the Landsat quality bit band to filter out clouds
raster_cube(L8.col, v, mask=image_mask("BQA", bits=4, values=16))
```

---

read\_chunk\_as\_array    *Read chunk data of a data cube from stdin or a file*

---

**Description**

This function can be used within function passed to [chunk\\_apply](#) in order to read a data cube chunk as a four-dimensional R array. It works only for R processes, which have been started from the gdalcubes C++ library. The resulting array has dimensions band, time, y, x (in this order).

**Usage**

```
read_chunk_as_array(with.dimnames = TRUE)
```

**Arguments**

`with.dimnames` if TRUE, the resulting array will contain dimnames with coordinates, datetime, and band names

**Value**

four-dimensional array

**Note**

Call this function ONLY from a function passed to [chunk\\_apply](#).

This function only works in R sessions started from gdalcubes streaming.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")

L8.cube = raster_cube(L8.col, v)
L8.cube = select_bands(L8.cube, c("B04", "B05"))
f <- function() {
  x <- read_chunk_as_array()
  out <- reduce_time(x, function(x) {
    cor(x[1,], x[2,], use="na.or.complete", method = "kendall")
  })
  write_chunk_from_array(out)
}
L8.cor = chunk_apply(L8.cube, f)
plot(L8.cor, zlim=c(0,1), key.pos=1)
```



---

reduce_space	<i>Reduce multidimensional data over space</i>
--------------	--

---

### Description

This generic function applies a reducer function over a data cube, an R array, or other classes if implemented.

### Usage

```
reduce_space(x, ...)
```

### Arguments

x	object to be reduced
...	further arguments passed to specific implementations

### Value

return value and type depend on the class of x

### See Also

[reduce\\_space.cube](#)  
[reduce\\_space.array](#)

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
reduce_space(raster_cube(L8.col, v) , "median(B02)")

d <- c(4,16,32,32)
x <- array(rnorm(prod(d)), d)
y <- reduce_space(x, function(v) {
  apply(v, 1, mean)
})
```

---

reduce_space.array	<i>Apply a function over space and bands in a four-dimensional (band, time, y, x) array and reduce spatial dimensions</i>
--------------------	---

---

### Description

Apply a function over space and bands in a four-dimensional (band, time, y, x) array and reduce spatial dimensions

### Usage

```
## S3 method for class 'array'
reduce_space(x, FUN, ...)
```

### Arguments

x	four-dimensional input array with dimensions band, time, y, x (in this order)
FUN	function which receives one spatial slice in a three-dimensional array with dimensions bands, y, x as input
...	further arguments passed to FUN

### Details

FUN is expected to produce a numeric vector (or scalar) where elements are interpreted as new bands in the result.

### Note

This is a helper function that uses the same dimension ordering as gdalcubes streaming. It can be used to simplify the application of R functions e.g. over spatial slices in a data cube.

### Examples

```
d <- c(4,16,32,32)
x <- array(rnorm(prod(d)), d)
# reduce individual bands over spatial slices
y <- reduce_space(x, function(v) {
  apply(v, 1, mean)
})
dim(y)
```

---

reduce\_space.cube      *Reduce a data cube over spatial (x,y or lat,lon) dimensions*

---

### Description

Create a proxy data cube, which applies one or more reducer functions to selected bands over spatial slices of a data cube

### Usage

```
## S3 method for class 'cube'
reduce_space(
  x,
  expr,
  ...,
  FUN,
  names = NULL,
  load_pkgs = FALSE,
  load_env = FALSE
)
```

### Arguments

x	source data cube
expr	either a single string, or a vector of strings defining which reducers will be applied over which bands of the input cube
...	optional additional expressions (if expr is not a vector)
FUN	a user-defined R function applied over pixel time series (see Details)
names	character vector; names of the output bands, if FUN is provided, the length of names is used as the expected number of output bands
load_pkgs	logical or character; if TRUE, all currently attached packages will be attached automatically before executing FUN in spawned R processes, specific packages can alternatively be provided as a character vector.
load_env	logical or environment; if TRUE, the current global environment will be restored automatically before executing FUN in spawned R processes, can be set to a custom environment.

### Details

Notice that expressions have a very simple format: the reducer is followed by the name of a band in parentheses. You cannot add more complex functions or arguments.

Possible reducers currently include "min", "max", "sum", "prod", "count", "mean", "median", "var", and "sd".

For more details and examples on how to write user-defined functions, please refer to the gdalcubes website at <https://gdalcubes.github.io/source/concepts/udfs.html>.

**Value**

proxy data cube object

**Note**

Implemented reducers will ignore any NAN values (as na.rm=TRUE does).

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")

L8.cube = raster_cube(L8.col, v)
L8.b02 = select_bands(L8.cube, c("B02"))
L8.b02.median = reduce_space(L8.b02, "median(B02)")
L8.b02.median

plot(L8.b02.median)
```

---

reduce\_time

*Reduce multidimensional data over time*

---

**Description**

This generic function applies a reducer function over a data cube, an R array, or other classes if implemented.

**Usage**

```
reduce_time(x, ...)
```

**Arguments**

x                    object to be reduced  
 ...                  further arguments passed to specific implementations

**Value**

return value and type depend on the class of x

**See Also**

[reduce\\_time.cube](#)

[reduce\\_time.array](#)

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-01", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
reduce_time(raster_cube(L8.col, v) , "median(B02)", "median(B03)", "median(B04)")

d <- c(4,16,32,32)
x <- array(rnorm(prod(d)), d)
y <- reduce_time(x, function(v) {
  apply(v, 1, mean)
})
```

---

reduce_time.array	<i>Apply a function over time and bands in a four-dimensional (band, time, y, x) array and reduce time dimension</i>
-------------------	--

---

**Description**

Apply a function over time and bands in a four-dimensional (band, time, y, x) array and reduce time dimension

**Usage**

```
## S3 method for class 'array'
reduce_time(x, FUN, ...)
```

**Arguments**

x	four-dimensional input array with dimensions band, time, y, x (in this order)
FUN	function which receives one time series in a two-dimensional array with dimensions bands, time as input
...	further arguments passed to FUN

**Details**

FUN is expected to produce a numeric vector (or scalar) where elements are interpreted as new bands in the result.

**Note**

This is a helper function that uses the same dimension ordering as gdalcubes streaming. It can be used to simplify the application of R functions e.g. over time series in a data cube.

**Examples**

```
d <- c(4,16,32,32)
x <- array(rnorm(prod(d)), d)
# reduce individual bands over pixel time series
y <- reduce_time(x, function(v) {
  apply(v, 1, mean)
})
dim(y)
```

---

reduce\_time.cube

*Reduce a data cube over the time dimension*

---

**Description**

Create a proxy data cube, which applies one or more reducer functions to selected bands over pixel time series of a data cube

**Usage**

```
## S3 method for class 'cube'
reduce_time(
  x,
  expr,
  ...,
  FUN,
  names = NULL,
  load_pkgs = FALSE,
  load_env = FALSE
)
```

## Arguments

x	source data cube
expr	either a single string, or a vector of strings defining which reducers will be applied over which bands of the input cube
...	optional additional expressions (if expr is not a vector)
FUN	a user-defined R function applied over pixel time series (see Details)
names	character vector; names of the output bands, if FUN is provided, the length of names is used as the expected number of output bands
load_pkgs	logical or character; if TRUE, all currently attached packages will be attached automatically before executing FUN in spawned R processes, specific packages can alternatively be provided as a character vector.
load_env	logical or environment; if TRUE, the current global environment will be restored automatically before executing FUN in spawned R processes, can be set to a custom environment.

## Details

The function can either apply a built-in reducer if expr is given, or apply a custom R reducer function if FUN is provided.

In the former case, notice that expressions have a very simple format: the reducer is followed by the name of a band in parentheses. You cannot add more complex functions or arguments. Possible reducers currently are "min", "max", "sum", "prod", "count", "mean", "median", "var", "sd", "which\_min", "which\_max", "Q1" (1st quartile), and "Q3" (3rd quartile).

User-defined R reducer functions receive a two-dimensional array as input where rows correspond to the band and columns represent the time dimension. For example, one row is the time series of a specific band. FUN should always return a numeric vector with the same number of elements, which will be interpreted as bands in the result cube. Notice that it is recommended to specify the names of the output bands as a character vector. If names are missing, the number and names of output bands is tried to be derived automatically, which may fail in some cases.

For more details and examples on how to write user-defined functions, please refer to the gdalcubes website at <https://gdalcubes.github.io/source/concepts/udfs.html>.

## Value

proxy data cube object

## Note

Implemented reducers will ignore any NAN values (as na.rm=TRUE does)

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```

# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-06"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
L8.cube = raster_cube(L8.col, v)
L8.rgb = select_bands(L8.cube, c("B02", "B03", "B04"))
L8.rgb.median = reduce_time(L8.rgb, "median(B02)", "median(B03)", "median(B04)")
L8.rgb.median

plot(L8.rgb.median, rgb=3:1)

# user defined reducer calculating interquartile ranges
L8.rgb.iqr = reduce_time(L8.rgb, names=c("iqr_R", "iqr_G", "iqr_B"), FUN = function(x) {
  c(diff(quantile(x[["B04"],],c(0.25,0.75), na.rm=TRUE)),
    diff(quantile(x[["B03"],],c(0.25,0.75), na.rm=TRUE)),
    diff(quantile(x[["B02"],],c(0.25,0.75), na.rm=TRUE)))
})
L8.rgb.iqr

plot(L8.rgb.iqr, key.pos=1)

```

---

**rename\_bands***Rename bands of a data cube*

---

**Description**

Create a proxy data cube, which renames specific bands of a data cube.

**Usage**

```
rename_bands(cube, ...)
```

**Arguments**

cube	source data cube
...	named arguments with bands that will be renamed, see Details



**Details**

The result data cube always contains the same number of bands. No subsetting is done if only names for some of the bands are provided. In this case, only provided bands are renamed whereas other bands keep their original name. Variable arguments must be named by the old band name and the new names must be provided as simple character values (see example).

**Value**

proxy data cube object

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-07"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
L8.cube = raster_cube(L8.col, v)
L8.rgb = select_bands(L8.cube, c("B02", "B03", "B04"))
L8.rgb
L8.rgb = rename_bands(L8.cube, B02 = "blue", B03 = "green", B04 = "red")
L8.rgb
```

---

 select\_bands

*Select bands of a data cube*


---

**Description**

Create a proxy data cube, which selects specific bands of a data cube. The resulting cube will drop any other bands.

**Usage**

```
select_bands(cube, bands)
```

**Arguments**

cube	source data cube
bands	character vector with band names

**Value**

proxy data cube object

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

For performance reasons, `select_bands` should always be called directly on a cube created with `raster_cube` and drop all unneeded bands. This allows to reduce RasterIO and warp operations in GDAL.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-07"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
L8.cube = raster_cube(L8.col, v)
L8.rgb = select_bands(L8.cube, c("B02", "B03", "B04"))
L8.rgb

plot(L8.rgb, rgb=3:1)
```

---

select\_time

*Select time slices of a data cube*

---

**Description**

Create a proxy data cube, which selects specific time slices of a data cube. The time dimension of the resulting cube will be irregular / labeled.

**Usage**

```
select_time(cube, t)
```

**Arguments**

cube	source data cube
t	character vector with date/time

**Value**

proxy data cube object

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-04", t1="2018-07"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
L8.cube = raster_cube(L8.col, v)
L8.rgb = select_bands(L8.cube, c("B02", "B03", "B04"))
L8.rgb = select_time(L8.rgb, c("2018-04", "2018-07"))
L8.rgb

plot(L8.rgb, rgb=3:1)
```

---

size

*Query data cube properties*

---

**Description**

Query data cube properties

**Usage**

size(obj)

**Arguments**

obj                    a data cube proxy object (class cube)

**Value**

size of a data cube (number of cells) as integer vector in the order t, y, x

**See Also**

[dim.cube](#)

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
size(raster_cube(L8.col, v))
```

---

slice\_space

*Extract a single time series (spatial slice) from a data cube*

---

**Description**

Create a proxy data cube, which extracts a time series from a data cube defined by spatial coordinates or integer x and y indexes.

**Usage**

```
slice_space(cube, loc = NULL, i = NULL)
```

**Arguments**

cube                    source data cube

loc                    numeric length-two vector; spatial coordinates (x, y) of the time series, expressed in the coordinate reference system of the source data cube

i                      integer length-2 vector; indexes (x,y) of the time slice (zero-based)

**Details**

Either loc or i must be non-NULL. If both arguments are provided, integer indexes i are ignored.

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}
L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P3M", aggregation = "median")
L8.cube = raster_cube(L8.col, v, mask=image_mask("BQA", bits=4, values=16))
L8.rgb = select_bands(L8.cube, c("B02", "B03", "B04"))
L8.ts = slice_space(L8.rgb, loc = c(5e05, 4400000))
L8.ts

plot(L8.ts, join.timeseries = TRUE)
```

---

slice\_time

*Extract a single time slice from a data cube*

---

**Description**

Create a proxy data cube, which extracts a time slice from a data cube defined by label (datetime string) or integer index.

**Usage**

```
slice_time(cube, datetime = NULL, it = NULL)
```

**Arguments**

cube	source data cube
datetime	character; datetime string of the time slice
it	integer; index of the time slice (zero-based)

**Details**

Either datetime or it must be non-NULL. If both arguments are provided, the integer index it is ignored.

**Note**

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}
L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M", aggregation = "median")
L8.cube = raster_cube(L8.col, v, mask=image_mask("BQA", bits=4, values=16))
L8.rgb = select_bands(L8.cube, c("B02", "B03", "B04"))
L8.slice = slice_time(L8.rgb, "2018-03")
L8.slice

plot(L8.slice, rgb=3:1, zlim=c(5000,12000))
```

---

srs

*Query data cube properties*


---

**Description**

Query data cube properties

**Usage**

```
srs(obj)
```

**Arguments**

obj                    a data cube proxy object (class cube)

**Value**

The spatial reference system expressed as a string readable by GDAL

## Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
srs(raster_cube(L8.col, v))
```

---

stack_cube	<i>Create a data cube from a set of images with the same spatial extent and spatial reference system</i>
------------	--

---

## Description

Create a spatiotemporal data cube directly from images with identical spatial extent and spatial reference system, similar to a raster stack with an additional dimension supporting both, time and multiple bands / variables.

## Usage

```
stack_cube(
  x,
  datetime_values,
  bands = NULL,
  band_names = NULL,
  chunking = c(1, 256, 256),
  dx = NULL,
  dy = NULL,
  incomplete_ok = TRUE
)
```

## Arguments

x	character vector where items point to image files
datetime_values	vector of type character, Date, or POSIXct with recording date of images
bands	optional character vector defining the band or spectral band of each item in x, if files relate to different spectral bands or variables
band_names	name of bands, only used if bands is NULL, i.e., if all files contain the same spectral band(s) / variable(s)

chunking	vector of length 3 defining the size of data cube chunks in the order time, y, x.
dx	optional target pixel size in x direction, by default (NULL) the original or highest resolution of images is used
dy	optional target pixel size in y direction, by default (NULL) the original or highest resolution of images is used
incomplete_ok	logical, if TRUE (the default), chunks will ignore IO failures and simply use as much images as possible, otherwise the result will contain empty chunks if IO errors or similar occur.

### Details

This function creates a four-dimensional (space, time, bands / variables) raster data cube from a set of provided files without the need to create an image collection before. This is possible if all images have the same spatial extent and spatial reference system and can be used for two different file organizations:

1. If all image files share the same bands / variables, the bands argument can be ignored (default NULL) can names of the bands can be specified using the band\_names argument.
2. If image files represent different band / variable (e.g. individual files for red, green, and blue channels), the bands argument must be used to define the corresponding band / variable. Notice that in this case all files are expected to represent exactly one variable / band at one point in datetime. It is not possible to combine files with different numbers of variables / bands. If image files for different bands have different pixel sizes, the smallest size is used by default.

Notice that to avoid opening all image files in advance, no automatic check whether all images share the spatial extent and spatial reference system is performed.

### Value

A proxy data cube object

### Note

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

### Examples

```
# toy example, repeating the same image as a daily time series
L8_file_nir <-
system.file("L8NY18/LC08_L1TP_014032_20181122_20181129_01_T1/LC08_L1TP_014032_20181122_B5.TIF",
            package = "gdalcubes")
files = rep(L8_file_nir, 10)
datetime = as.Date("2018-11-22") + 1:10
stack_cube(files, datetime, band_names = "B05")

# using a second band from different files
L8_file_red <-
system.file("L8NY18/LC08_L1TP_014032_20181122_20181129_01_T1/LC08_L1TP_014032_20181122_B4.TIF",
            package = "gdalcubes")
files = rep(c(L8_file_nir, L8_file_red), each = 10)
```



```

datetime = rep(as.Date("2018-11-22") + 1:10, 2)
bands = rep(c("B5", "B4"), each = 10)
stack_cube(files, datetime, bands = bands)

```

---

stac\_image\_collection *Create an image collection from a STAC feature collection*

---

## Description

This function creates an image collection from a STAC API collection response. It does not need to read any image data. Additionally, bands can be filtered and asset links can be transformed to make them readable for GDAL.

## Usage

```

stac_image_collection(
  s,
  out_file = tempfile(fileext = ".db"),
  asset_names = NULL,
  asset_regex = NULL,
  url_fun = .default_url_fun,
  property_filter = NULL,
  skip_image_metadata = FALSE,
  srs = NULL,
  srs_overwrite = FALSE,
  duration = c("center", "start")
)

```

## Arguments

s	STAC feature collection
out_file	optional name of the output SQLite database file, defaults to a temporary file
asset_names	character vector with names of assets (e.g., bands) to be used, other assets will be ignored. By default (NULL), all asset names with "eo:bands" attributes will be used
asset_regex	length 1 character defining a regular expression asset names must match to be considered
url_fun	optional function to modify URLs of assets, e.g, to add /vsicurl/ to URLs (the default)
property_filter	optional function to filter STAC items (images) by their properties; see Details
skip_image_metadata	logical, if TRUE per-image metadata (STAC item properties) will not be added to the image collection

srs	character spatial reference system of images used either for images without corresponding STAC property only or for all images
srs_overwrite	logical, if FALSE, use srs only for images with unknown srs (missing STAC metadata)
duration	character, if images represent time intervals, use either the "start" or "center" of time intervals

### Details

The `property_filter` argument can be used to filter images by metadata such as cloud coverage. The functions receives all properties of a STAC item (image) as input list and is expected to produce a single logical value, where an image will be ignored if the function returns FALSE.

Some STAC API endpoints may return items with duplicate IDs (image names), pointing to identical URLs. Such items are only added once during creation of the image collection.

### Note

Currently, `bbox` results are expected to be WGS84 coordinates, even if `bbox-crs` is given in the STAC response.

---

st_as_stars.cube	<i>Coerce gdalcubes object into a stars object</i>
------------------	--

---

### Description

The function materializes a data cube as a temporary netCDF file and loads the file with the stars package.

### Usage

```
st_as_stars.cube(.x, ...)
```

### Arguments

.x	data cube object to coerce
...	not used

### Value

stars object

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-04", t1="2018-04"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
if(require("stars"))
  st_as_stars(select_bands(raster_cube(L8.col, v), c("B04", "B05")))
```

---

window_space	<i>Apply a moving window (focal) operation or a convolution kernel over spatial dimensions of a data cube.</i>
--------------	--

---

**Description**

Create a proxy data cube, which applies a convolution kernel or aggregation functions over two-dimensional moving windows sliding over spatial slices of a data cube. The function can either execute one or more predefined aggregation functions or apply a custom convolution kernel. Among others, use cases include image processing (edge detection, noise reduction, etc.) and enriching pixel values with local neighborhood properties (e.g. to use as predictor variables in ML models).

**Usage**

```
window_space(x, expr, ..., kernel, window, keep_bands = FALSE, pad = NA)
```

**Arguments**

x	source data cube
expr	either a single string, or a vector of strings, defining which reducers will be applied over which bands of the input cube
...	optional additional expressions (if expr is not a vector)
kernel	two dimensional kernel (matrix) applied as convolution (with odd number of rows and columns)
window	integer vector with two elements defining the size (number of pixels) of the window in y and x direction, the total size of the window is window[1] * window[2]
keep_bands	logical; if FALSE (the default), original data cube bands will be dropped.
pad	padding method applied to the borders; use NULL for no padding (NA), a numeric a fill value, or one of "REPLICATE", "REFLECT", "REFLECT_PIXEL"

## Details

The function either applies a kernel convolution (if the kernel argument is provided) or one or more built-in reducer function over moving windows.

In the former case, the kernel convolution will be applied over all bands of the input cube, i.e., the output cube will have the same number of bands as the input cubes.

To apply one or more aggregation functions over moving windows, the window argument must be provided as a vector with two integer sizes in the order y, x. Several string expressions can be provided to create multiple bands in the output cube. Notice that expressions have a very simple format: the reducer is followed by the name of a band in parentheses, e.g, "mean(band1)". Possible reducers include "min", "max", "sum", "prod", "count", "mean", "median", "var", and "sd".

Padding methods "REPLICATE", "REFLECT", "REFLEX\_PIXEL" are defined according to [https://openeo.org/documentation/1.0/processes.html#apply\\_kernel](https://openeo.org/documentation/1.0/processes.html#apply_kernel).

## Value

proxy data cube object

## Note

Implemented reducers will ignore any NAN values (as `na.rm = TRUE` does).

Calling this function consecutively many times may result in long computation times depending on chunk and window sizes due to the need to read adjacent data cube chunks.

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

## Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-04", t1="2018-06"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
L8.cube = raster_cube(L8.col, v, chunking = c(1,1000,1000))
L8.cube = select_bands(L8.cube, c("B04", "B05"))
L8.cube.mean5x5 = window_space(L8.cube, kernel = matrix(1/25, 5, 5))
L8.cube.mean5x5

plot(L8.cube.mean5x5, key.pos=1)

L8.cube.med_sd = window_space(L8.cube, "median(B04)", "sd(B04)", "median(B05)", "sd(B05)",
```

```

                                window = c(5,5), keep_bands = TRUE)
L8.cube.med_sd
plot(L8.cube.med_sd, key.pos=1)

```

---

window_time	<i>Apply a moving window operation over the time dimension of a data cube</i>
-------------	---

---

### Description

Create a proxy data cube, which applies one or more moving window functions to selected bands over pixel time series of a data cube. The function can either apply a built-in aggregation function or apply a custom one-dimensional convolution kernel.

### Usage

```
window_time(x, expr, ..., kernel, window)
```

### Arguments

x	source data cube
expr	either a single string, or a vector of strings defining which reducers will be applied over which bands of the input cube
...	optional additional expressions (if expr is not a vector)
kernel	numeric vector with elements of the kernel
window	integer vector with two elements defining the size of the window before and after a cell, the total size of the window is window[1] + 1 + window[2]

### Details

The function either applies a kernel convolution (if the kernel argument is provided) or a general reducer function over moving temporal windows. In the former case, the kernel convolution will be applied over all bands of the input cube, i.e., the output cube will have the same number of bands as the input cubes. If a kernel is given and the window argument is missing, the window will be symmetric to the center pixel with the size of the provided kernel. For general reducer functions, the window argument must be provided and several expressions can be used to create multiple bands in the output cube.

Notice that expressions have a very simple format: the reducer is followed by the name of a band in parentheses. You cannot add more complex functions or arguments.

Possible reducers include "min", "max", "sum", "prod", "count", "mean", and "median".

### Value

proxy data cube object

**Note**

Implemented reducers will ignore any NAN values (as na.rm=TRUE does).

This function returns a proxy object, i.e., it will not start any computations besides deriving the shape of the result.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-01", t1="2018-07"),
  srs="EPSG:32618", nx = 400, dt="P1M")
L8.cube = raster_cube(L8.col, v)
L8.nir = select_bands(L8.cube, c("B05"))
L8.nir.min = window_time(L8.nir, window = c(2,2), "min(B05)")
L8.nir.min

L8.nir.kernel = window_time(L8.nir, kernel=c(-1,1), window=c(1,0))
L8.nir.kernel
```

---

```
write_chunk_from_array
```

*Write chunk data of a cube to stdout or a file*

---

**Description**

This function can be used within function passed to [chunk\\_apply](#) in order to pass four-dimensional R arrays as a data cube chunk to the gdalcubes C++ library. It works only for R processes, which have been started from the gdalcubes C++ library. The input array must have dimensions band, time, y, x (in this order).

**Usage**

```
write_chunk_from_array(v)
```

**Arguments**

v                    four-dimensional array with dimensions band, time, y, and x

**Note**

Call this function **ONLY** from a function passed to `chunk_apply`.

This function only works in R sessions started from gdalcubes streaming.

**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                          bottom=4345299, top=4744931, t0="2018-01", t1="2018-12"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")

L8.cube = raster_cube(L8.col, v)
L8.cube = select_bands(L8.cube, c("B04", "B05"))
f <- function() {
  x <- read_chunk_as_array()
  out <- reduce_time(x, function(x) {
    cor(x[1,], x[2,], use="na.or.complete", method = "kendall")
  })
  write_chunk_from_array(out)
}
L8.cor = chunk_apply(L8.cube, f)
plot(L8.cor, zlim=c(0,1), key.pos=1)
```

---

write\_ncdf

*Export a data cube as netCDF file(s)*


---

**Description**

This function will read chunks of a data cube and write them to a single (the default) or multiple (if `chunked = TRUE`) netCDF file(s). The resulting file(s) uses the enhanced netCDF-4 format, supporting chunking and compression.

**Usage**

```
write_ncdf(
  x,
  fname = tempfile(pattern = "gdalcubes", fileext = ".nc"),
  overwrite = FALSE,
  write_json_descr = FALSE,
  with_VRT = FALSE,
```

```

    pack = NULL,
    chunked = FALSE
)

```

### Arguments

x	a data cube proxy object (class cube)
fname	output file name
overwrite	logical; overwrite output file if it already exists
write_json_descr	logical; write a JSON description of x as additional file
with_VRT	logical; write additional VRT datasets (one per time slice)
pack	reduce output file size by packing values (see Details), defaults to no packing
chunked	logical; if TRUE, write one netCDF file per chunk; defaults to FALSE

### Details

The resulting netCDF file(s) contain three dimensions (t, y, x) and bands as variables.

If `write_json_descr` is TRUE, the function will write an addition file with the same name as the NetCDF file but ".json" suffix. This file includes a serialized description of the input data cube, including all chained data cube operations.

To reduce the size of created files, values can be packed by applying a scale factor and an offset value and using a smaller integer data type for storage (only supported if `chunked = TRUE`). The `pack` argument can be either NULL (the default), or a list with elements `type`, `scale`, `offset`, and `nodata`. `type` can be any of "uint8", "uint16", "uint32", "int16", or "int32". `scale`, `offset`, and `nodata` must be numeric vectors with length one or length equal to the number of data cube bands (to use different values for different bands). The helper function [pack\\_minmax](#) can be used to derive offset and scale values with maximum precision from minimum and maximum data values on original scale.

If `chunked = TRUE`, names of the produced files will start with name (with removed extension), followed by an underscore and the internal integer chunk number.

### Value

returns (invisibly) the path of the created netCDF file(s)

### Note

Packing is currently ignored if `chunked = TRUE`

### See Also

[gdalcubes\\_options](#)  
[pack\\_minmax](#)



**Examples**

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
    ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
  bottom=4345299, top=4744931, t0="2018-04", t1="2018-04"),
  srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
write_ncdf(select_bands(raster_cube(L8.col, v), c("B04", "B05")), fname=tempfile(fileext = ".nc"))
```

write\_tif

*Export a data cube as a collection of GeoTIFF files***Description**

This function will time slices of a data cube as GeoTIFF files in a given directory.

**Usage**

```
write_tif(
  x,
  dir = tempfile(pattern = ""),
  prefix = basename(tempfile(pattern = "cube_")),
  overviews = FALSE,
  COG = FALSE,
  rsmpl_overview = "nearest",
  creation_options = NULL,
  write_json_descr = FALSE,
  pack = NULL
)
```

**Arguments**

x	a data cube proxy object (class cube)
dir	destination directory
prefix	output file name
overviews	logical; generate overview images
COG	logical; create cloud-optimized GeoTIFF files (forces overviews=TRUE)
rsmpl_overview	resampling method for overviews (image pyramid) generation (see <a href="https://gdal.org/programs/gdaladdo.html">https://gdal.org/programs/gdaladdo.html</a> for available methods)

creation_options	additional creation options for resulting GeoTIFF files, e.g. to define compression (see <a href="https://gdal.org/drivers/raster/gtiff.html#creation-options">https://gdal.org/drivers/raster/gtiff.html#creation-options</a> )
write_json_descr	logical; write a JSON description of x as additional file
pack	reduce output file size by packing values (see Details), defaults to no packing

### Details

If `write_json_descr` is TRUE, the function will write an additional file with name according to prefix (if not missing) or simply `cube.json`. This file includes a serialized description of the input data cube, including all chained data cube operations.

Additional GDAL creation options for resulting GeoTIFF files must be passed as a named list of simple strings, where element names refer to the key. For example, `creation_options = list("COMPRESS" = "DEFLATE", "ZLEVEL" = "5")` would enable deflate compression at level 5.

To reduce the size of created files, values can be packed by applying a scale factor and an offset value and using a smaller integer data type for storage. The `pack` argument can be either NULL (the default), or a list with elements `type`, `scale`, `offset`, and `nodata`. `type` can be any of "uint8", "uint16", "uint32", "int16", or "int32". `scale`, `offset`, and `nodata` must be numeric vectors with length one or length equal to the number of data cube bands (to use different values for different bands). The helper function `pack_minmax` can be used to derive offset and scale values with maximum precision from minimum and maximum data values on original scale.

If `overviews=TRUE`, the numbers of pixels are halved until the longer spatial dimensions counts less than 256 pixels. Setting `COG=TRUE` automatically sets `overviews=TRUE`.

### Value

returns (invisibly) a vector of paths pointing to the created GeoTIFF files

### See Also

[pack\\_minmax](#)

### Examples

```
# create image collection from example Landsat data only
# if not already done in other examples
if (!file.exists(file.path(tempdir(), "L8.db"))) {
  L8_files <- list.files(system.file("L8NY18", package = "gdalcubes"),
                        ".TIF", recursive = TRUE, full.names = TRUE)
  create_image_collection(L8_files, "L8_L1TP", file.path(tempdir(), "L8.db"), quiet = TRUE)
}

L8.col = image_collection(file.path(tempdir(), "L8.db"))
v = cube_view(extent=list(left=388941.2, right=766552.4,
                        bottom=4345299, top=4744931, t0="2018-04", t1="2018-04"),
              srs="EPSG:32618", nx = 497, ny=526, dt="P1M")
write_tif(select_bands(raster_cube(L8.col, v), c("B04", "B05")), dir=tempdir())
```

# Index

.copy\_cube, 4  
[.cube (gdalcubes\_selection), 42  
\$.cube (gdalcubes\_selection), 42

add\_collection\_format, 4  
add\_images, 5  
aggregate\_space, 6  
aggregate\_time, 7  
animate, 8  
apply\_pixel, 10  
apply\_pixel.array, 10, 11  
apply\_pixel.cube, 10, 12  
apply\_time, 14  
apply\_time.array, 14, 15  
apply\_time.cube, 14, 16  
as.data.frame.cube, 18  
as\_array, 19  
as\_json, 20, 48

bands, 21

chunk\_apply, 21, 63, 64, 86, 87  
collection\_formats, 23, 24  
create\_image\_collection, 23, 45  
crop, 25  
cube\_view, 27

dim.cube, 29, 76  
dimension\_bounds, 31  
dimension\_values, 31  
dimensions, 30

extent, 27, 28, 32  
extract\_geom, 33

fill\_time, 35  
filter\_geom, 36  
filter\_pixel, 38

gdalcubes, 39  
gdalcubes\_gdal\_has\_geos, 40

gdalcubes\_gdalformats, 39  
gdalcubes\_gdalversion, 40  
gdalcubes\_options, 40, 57, 88  
gdalcubes\_selection, 42  
gdalcubes\_set\_gdal\_config, 44

image\_collection, 45  
image\_mask, 45, 62

join\_bands, 47  
json\_cube, 48, 48

memsize, 49  
merge(), 34

names.cube, 50  
nbands, 50  
ncdf\_cube, 51  
nt, 52  
nx, 53  
ny, 54

pack\_minmax, 54, 88, 90  
plot.cube, 9, 55  
predict.cube, 58  
print.cube, 59  
print.cube\_view, 60  
print.image\_collection, 61  
proj4, 61

raster\_cube, 5, 22, 25, 45, 48, 62, 74  
read\_chunk\_as\_array, 63  
reduce\_space, 65  
reduce\_space.array, 65, 66  
reduce\_space.cube, 65, 67  
reduce\_time, 16, 68  
reduce\_time.array, 69, 69  
reduce\_time.cube, 69, 70  
rename\_bands, 72  
select\_bands, 73

select\_time, 74  
sf package, 33  
size, 29, 75  
slice\_space, 76  
slice\_time, 77  
srs, 78  
st\_as\_stars.cube, 82  
st\_bbox, 42  
st\_sfc, 42  
stac\_image\_collection, 81  
stack\_cube, 79  
  
window\_space, 83  
window\_time, 85  
write\_chunk\_from\_array, 86  
write\_ncdf, 51, 54, 87  
write\_tif, 54, 89