

Package: tilework (via r-universe)

October 8, 2026

Title Extensible Tile Planning and Windowed Parallel Processing

Version 1.0.0

Description An extensible framework for tile-based processing of large, spatially indexed data. Tile plans are first-class objects describing a set of bounds, which may be regular grids in pixel or coordinate space, tiles centered on supplied points, or adaptive decompositions such as quadtrees. Bounds may carry padding, supplying the overlapping context that neighborhood operations and objects spanning several tiles require. Extraction dispatches on the class of the input, so one plan can window two data sources of differing type together; methods are provided for 'terra' rasters and vector proxies, and other packages can register methods for their own formats. Tiles are processed in parallel through 'future', or optionally 'BiocParallel', and a per-worker initialization step amortizes expensive setup such as loading a model.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

Depends R (>= 4.1.0)

Imports checkmate, future, future.apply, methods, progressr, terra (>= 1.8-21)

Suggests abind, BiocParallel, knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

URL <https://drieslab.github.io/tilework/>,
<https://github.com/drieslab/tilework>

VignetteBuilder knitr

BugReports <https://github.com/drieslab/tilework/issues>

Collate 'classes.R' 'generics.R' 'package_imports.R' 'tilePlan.R'
'freeTilePlan.R' 'getBoundedData.R' 'pointTilePlan.R'
'getTile.R' 'intersect.R' 'parallel.R' 'pixelTilePlan.R'

```
'spatialTilePlan.R' 'tileGroup.R' 'tileApply.R'
'tileApply_tileGroup.R' 'tileApply_tileIterator.R'
'tileApply_tilePlan.R' 'tileApply_tileSelection.R'
'tileIterator.R' 'tileSelection.R' 'utils-logging.R' 'utils.R'
```

Config/roxygen2/version 8.0.0

Config/pak/sysreqs libgdal-dev gdal-bin libgeos-dev libproj-dev libsqlite3-dev

Repository <https://drieslab.r-universe.dev>

Date/Publication 2026-09-17 06:53:43 UTC

RemoteUrl <https://github.com/drieslab/tilework>

RemoteRef HEAD

RemoteSha f4616fa3e803b1def473f68b0d8c296d86cb954a

Contents

arith	3
as.polygons	4
bracket	5
c	8
centroids	8
dim	9
dollar	10
double_bracket	12
ext	13
extending_tilework	14
freeTilePlan	15
freeTilePlan-class	15
getBoundedData	17
getTile	18
intersect	20
iterSplit	22
parallel_params	22
pixelTilePlan-class	24
plot	26
pointTilePlan-class	27
quadtreePlan	30
redispatch_tileapply,SpatRaster,freeTilePlan-method	31
spatialTilePlan-class	36
tileApply-group	38
tileApply-iterator	41
tileApply-plan	44
tileGroup	47
tileGroup-class	49
tileIterator	49
tileIterator-class	52
tilePlan	53
tilePlan-class	54

tilework-class	54
tilework_management	55
token-class	55

Index	57
--------------	-----------

arith	<i>Tile Pads</i>
-------	------------------

Description

Tile padding extends the bounds of the tile beyond the region that they are initially planned for by an equal amount on all 4 sides. This can help with things like spatial contiguity, so that artificial borders are less obvious.

Padding can be added either via `$pad` or `+` and `-` operators. The `+` and `-` operators specifically modify the padding value based on the arithmetic ops. This is similar to their usage in [terra::Arith-methods](#) `$stride<-` is an alternative way to set padding via the stride between tile starts: `pad = (tile_dim - stride) / 2`. Requires `tile_dims` to be set (i.e. not `freeTilePlan`). `$stride` returns the effective stride given the current `pad`.

Usage

```
## S4 method for signature 'tilePlan,numeric'
e1 + e2

## S4 method for signature 'tilePlan,numeric'
e1 - e2

## S4 method for signature 'pointTilePlan,numeric'
e1 + e2

## S4 method for signature 'pixelTilePlan,numeric'
e1 + e2

## S4 method for signature 'tileGroup,numeric'
e1 + e2

## S4 method for signature 'tileGroup,numeric'
e1 - e2

## S4 method for signature 'tileSelection,numeric'
e1 + e2

## S4 method for signature 'tileSelection,numeric'
e1 - e2
```

Arguments

e1 tilePlan
 e2 numeric to add to padding value

Value

tilePlan

spatial and pixel differences

- Padding for spatial tile plans is added without any further changes.
- Padding for pixel tile plans are automatically zeroed out against the top left

See Also

Other tile* methods: [as.polygons](#), [bracket](#), [centroids](#), [dim](#), [dollar](#), [double_bracket](#), [ext](#), [intersect](#), [plot](#)

Examples

```
spat <- tilePlan("spatial")
ext(spat) <- c(0, 1000, 0, 1000)
length(spat) <- 9
plot(spat)
spat[1:3]
spat <- spat + 10
plot(spat)
spat[1:3]

px <- tilePlan("pixel")
px$pxdims <- c(1000, 1000)
px$ncols <- 100
px$nrows <- 100
plot(px)
px[1:3]
px <- px + 5
plot(px)
px[1:3]
```

as.polygons

Coerce a tile plan to polygons

Description

Convert a tilePlan-inheriting object to a SpatVector of rectangle polygons, one per tile, with a tile attribute column holding the tile index. Padding is included in the polygon bounds.

Accelerated vectorized methods are provided for spatialTilePlan, freeTilePlan, and pointTilePlan. All other tilePlan subclasses fall back to extracting bounds via x[[]].

Usage

```
## S4 method for signature 'tilePlan'
as.polygons(x, ...)

## S4 method for signature 'freeTilePlan'
as.polygons(x, ...)

## S4 method for signature 'pointTilePlan'
as.polygons(x, ...)

## S4 method for signature 'spatialTilePlan'
as.polygons(x, ...)
```

Arguments

```
x          tilePlan-inheriting object
...        additional arguments (ignored)
```

Value

SpatVector of polygons

See Also

Other tile* methods: [arith](#), [bracket](#), [centroids](#), [dim](#), [dollar](#), [double_bracket](#), [ext](#), [intersect](#), [plot](#)

bracket	<i>Extract Bounds from Tile Object</i>
---------	--

Description

Get a set of tile bounds from a tile* object. Values are always returned as a list, even when length one to reduce surprises with lapply() usage.

Usage

```
## S4 method for signature 'tilePlan,numeric,missing,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'tilePlan,missing,numeric,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'tilePlan,numeric,numeric,missing'

x[
i,
```

```

    j,
    tile_fun = .spat_tile_bounds,
    fun = function(x) x,
    zero = FALSE,
    expand_grid = TRUE,
    drop
]

## S4 method for signature 'tilePlan,missing,missing,missing'
x[i, j]

## S4 method for signature 'tilePlan,numeric,missing,logical'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'tilePlan,missing,numeric,logical'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'tilePlan,numeric,numeric,logical'
x[i, j, expand_grid = TRUE, ..., drop]

## S4 method for signature 'tilePlan,missing,missing,logical'
x[i, j, drop]

## S4 method for signature 'freeTilePlan,numeric,numeric,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'pointTilePlan,numeric,missing,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'pixelTilePlan,numeric,numeric,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'spatialTilePlan,numeric,numeric,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'tileGroup,missing,missing,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'tileGroup,.index,missing,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'tileGroup,missing,.index,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'tileGroup,numeric,.index,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'tileIterator,missing,missing,missing'

```

```

x[i, j, ..., drop = TRUE]

## S4 replacement method for signature 'tileIterator,missing,missing'
x[i, j, ...] <- value

## S4 method for signature 'tileSelection,numeric,missing,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'tileSelection,numeric,missing,logical'
x[i, j, ..., drop = TRUE]

```

Arguments

x	tile* object
i	numeric vector index if j is not given. Row index if j is also present. Works like matrix-like indexing.
j	numeric. Column index
...	additional params to pass (not used).
drop	not used.
tile_fun	(internal) function used to define tile bounds based on the iterator object and ij indices.
fun	(internal) post-processing function used to convert bounds to expected output. Such as <code>ext()</code> for <code>spatialTilePlan</code>
zero	logical. Whether to zero out padding effects. (used by default with <code>pixelTilePlan</code>)
expand_grid	logical (internal) whether to use <code>expand.grid()</code> on ij indices.
value	ANY value to set

Value

list of numeric or `SpatExtent` depending on underlying `tilePlan`

See Also

Other tile* methods: [arith](#), [as.polygons](#), [centroids](#), [dim](#), [dollar](#), [double_bracket](#), [ext](#), [intersect](#), [plot](#)

Examples

```

spat <- tilePlan("spatial")
ext(spat) <- c(0, 1000, 0, 1000)
length(spat) <- 9
spat[1]
spat[1, 2:3]
spat[]

px <- tilePlan("pixel")
px$pxdims <- c(1000, 1000)
px$ncols <- 200

```

```
px$nrows <- 200
px[1]
px[2, 2:3]
px[]
```

c

*Combine tileSelections***Description**

Combine tileSelection. Should only concatenate tileSelection build off the same plan.

Usage

```
## S4 method for signature 'tileSelection'
c(x, ...)
```

Arguments

x	item to concatenate
...	additional items to concatenate to x

centroids

*Get Tile Centroids***Description**

Get the centroids of the tiles. For spatialTilePlan, these will be returned as SpatVector centroids. For non-spatial plans like pixelTileGrid, this will be returned as a matrix.

Usage

```
## S4 method for signature 'tilePlan'
centroids(x, fun = function(x) x, offset = c(0, 0), zero = FALSE, ...)

## S4 method for signature 'freeTilePlan'
centroids(x, ...)

## S4 method for signature 'pointTilePlan'
centroids(x, ...)

## S4 method for signature 'pixelTilePlan'
centroids(x, fun = function(x) x, zero = TRUE, ...)

## S4 method for signature 'spatialTilePlan'
centroids(x, zero = FALSE, ...)
```

Arguments

x	tilePlan
fun	(internal) post-processing function used to convert centroids to expected output. Such as vect() for spatialTilePlan
offset	numeric of length 2. x and y values to offset by. Default is c(0, 0).
zero	logical. Whether to zero out padding effects. (used by default with pixelTilePlan)
...	additional params to pass.

See Also

Other tile* methods: [arith](#), [as.polygons](#), [bracket](#), [dim](#), [dollar](#), [double_bracket](#), [ext](#), [intersect](#), [plot](#)

Examples

```
x <- tilePlan()
ext(x) <- c(0, 100, 0, 100)
length(x) <- 9
centroids(x)
```

dim

Tile Plan Array Characteristics

Description

Get dimension characteristics of the tilePlan tiling plan. These produce information on how the tiles are arrayed in rows nrow(), cols ncol(), and total number of tiles length().

length<-() is a special case and currently only used with spatialTilePlan, where it is used to request a minimum number of tiles to generate.

Usage

```
## S4 method for signature 'tilePlan'
nrow(x)

## S4 method for signature 'tilePlan'
ncol(x)

## S4 method for signature 'tilePlan'
length(x)

## S4 method for signature 'tilePlan'
dim(x)

## S4 replacement method for signature 'spatialTilePlan'
length(x) <- value
```

```
## S4 method for signature 'tileGroup'
length(x)

## S4 method for signature 'tileIterator'
length(x)

## S4 method for signature 'tileSelection'
length(x)
```

Arguments

x	tilePlan object
value	number of tiles to request

Value

numeric or spatialTilePlan object when using length<-()

See Also

Other tile* methods: [arith](#), [as.polygons](#), [bracket](#), [centroids](#), [dollar](#), [double_bracket](#), [ext](#), [intersect](#), [plot](#)

Examples

```
x <- tilePlan()
ext(x) <- c(0, 100, 0, 100)
length(x)
length(x) <- 9
length(x)
dim(x)
nrow(x)
ncol(x)
```

dollar

Get and Set Tile Metadata and Params

Description

Get and set tile metadata. Some params can also be modified with this operator, for example the \$pad value or \$pxdims, \$ncols, or \$nrows for pixelTilePlan. \$tile is a default created piece of metadata.

Usage

```
## S4 replacement method for signature 'tilePlan'
x$name <- value

## S4 method for signature 'tilePlan'
x$name

## S4 replacement method for signature 'freeTilePlan'
x$name <- value

## S4 method for signature 'freeTilePlan'
x$name

## S4 replacement method for signature 'pointTilePlan'
x$name <- value

## S4 method for signature 'pointTilePlan'
x$name

## S4 replacement method for signature 'pixelTilePlan'
x$name <- value

## S4 method for signature 'pixelTilePlan'
x$name

## S4 replacement method for signature 'tileGroup'
x$name <- value

## S4 method for signature 'tileGroup'
x$name

## S4 method for signature 'tileSelection'
x$name

## S4 replacement method for signature 'tileSelection'
x$name <- value
```

Arguments

x	tilePlan
name	name of item to get or set
value	value to set

Value

metadata value when using getter or tile* object when using the setter function.

See Also

Other `tile*` methods: [arith](#), [as.polygons](#), [bracket](#), [centroids](#), [dim](#), [double_bracket](#), [ext](#), [intersect](#), [plot](#)

Examples

```
x <- tilePlan()
ext(x) <- c(0, 100, 0, 100)
length(x) <- 9
x$test <- sprintf("test_value_%d", x$tile)
x$test
tile_ext <- x[5][[1]]
attr(tile_ext, "test")
```

double_bracket	<i>Get and set metadata</i>
----------------	-----------------------------

Description

`[[` can be used to get the table of metadata for a specific tile.

Usage

```
## S4 method for signature 'tilePlan,numeric,missing'
x[[i, j, ...]]

## S4 method for signature 'tilePlan,missing,missing'
x[[i, j, ...]]

## S4 method for signature 'tilePlan,missing,character'
x[[i, j, ...]]

## S4 method for signature 'tilePlan,numeric,character'
x[[i, j, ...]]

## S4 replacement method for signature 'tilePlan,numeric,character'
x[[i, j, ...]] <- value

## S4 replacement method for signature 'tilePlan,missing,character'
x[[i, j, ...]] <- value
```

Arguments

<code>x</code>	<code>tilePlan</code>
<code>i</code>	integer-like. tile vector index
<code>j</code>	character. Name of metadata information to get
<code>...</code>	not used.
<code>value</code>	ANY value to set

Value

data.frame

See Also

[dollar](#)

Other tile* methods: [arith](#), [as.polygons](#), [bracket](#), [centroids](#), [dim](#), [dollar](#), [ext](#), [intersect](#), [plot](#)

Examples

```
spat <- tilePlan("spatial")
ext(spat) <- c(0, 1000, 0, 1000)
length(spat) <- 9
spat[[1]]
```

ext

Get and Set Spatial Extent

Description

Get and set a spatial extent.

Usage

```
## S4 method for signature 'pointTilePlan'
ext(x, ...)

## S4 replacement method for signature 'pointTilePlan,ANY'
ext(x) <- value

## S4 method for signature 'spatialTilePlan'
ext(x, ...)

## S4 replacement method for signature 'spatialTilePlan,ANY'
ext(x) <- value
```

Arguments

x	tilePlan
...	additional params to pass (none implemented)
value	numeric of length 4 or SpatExtent defining spatial extent.

Value

tilePlan if ext<-() and SpatExtent if ext()

See Also

Other tile* methods: [arith](#), [as.polygons](#), [bracket](#), [centroids](#), [dim](#), [dollar](#), [double_bracket](#), [intersect](#), [plot](#)

extending_tilework	<i>Extending {tilework}</i>
--------------------	-----------------------------

Description

{tilework} provides an extensible framework for:

- [tile planning](#)
- [tile selection](#)
- [batch iteration](#)
- [parallelized data processing](#)

Data flow

The framework accesses and processes data according to the following flow:

1. `tileApply(x, tiles, ...)` # operate across tiles
2. `getTile(x, tiles, ..., get_params)` # get data for tiles
 - (a) `bounds <- tiles[...]` # generate bounds info
 - (b) `getBoundedData(x, bounds, get_params)` # data extraction

Expanded roles:

- [tileApply\(\)](#) - handles the function parallelization strategy and logic.
- [getTile\(\)](#) - deals with any interactions between the data type and the tile indexing. Operations that should be performed before extraction of the actual data by tile bounds should happen at this level.
- [getBoundedData\(\)](#) - performs the low-level data extraction from the data source, provided with a set of bounds information (commonly a `SpatExtent` or numeric of `c(xmin, xmax, ymin, ymax)`)

Adding new data type support

{tilework} functionalities can be extended to work with other data types/backends by creating a [getBoundedData\(\)](#) method for the data type. Optional additional steps:

- Method for [tileApply\(\)](#) should also be added if there are pre-run validation/data collection steps needed.
- Method for [getTile\(\)](#) should be added if there are operations that affect data/tile interactions.

See Also

Other tilework extension: [redispatch_tileapply](#), [SpatRaster](#), [freeTilePlan-method](#), [token-class](#)

Examples

```

if (FALSE) {
  # Extending for a custom data type:
  setMethod(
    "getBoundedData", signature("MyDataType", "SpatExtent"),
    function(x, bound) {
      # Your data extraction logic here
      my_extract_function(x, bound)
    }
  )
}

```

freeTilePlan

Create a Fluid Tile Plan

Description

Create an empty freeTilePlan. Populate bounds via `$bounds<-` or use [quadtreePlan\(\)](#) to generate adaptive bounds automatically.

Usage

```
freeTilePlan(...)
```

Arguments

... additional params passed to `new()`.

See Also

[quadtreePlan\(\)](#), [freeTilePlan](#)

Other tile plans: [freeTilePlan-class](#), [pixelTilePlan-class](#), [pointTilePlan-class](#), [quadtreePlan\(\)](#), [spatialTilePlan-class](#), [tilePlan\(\)](#), [tilePlan-class](#), [tilework-class](#)

freeTilePlan-class

Free Tile Plan

Description

Tile plan defined by explicit per-tile bounds with no required uniformity in size or spacing. The bounds matrix is the canonical representation — there is no center-plus-dims formula. Always returns `SpatExtent` from `[i]`. Primary use case is as the output of adaptive spatial decomposition algorithms such as quadtrees on vector/point data.

Tile plan defined by explicit per-tile bounds with no required uniformity in size or spacing. Unlike grid-based plans, the bounds matrix is the canonical representation — tile positions are not computed from a formula.

The primary use case is adaptive spatial decomposition of vector/point data, where high-density regions require small tiles and sparse regions can use large tiles. See [quadtreePlan\(\)](#) for the iterative construction workflow.

Slots

`bounds` matrix. `n x 4` matrix: `xmin`, `xmax`, `ymin`, `ymax` (one row per tile).

Setup

- `freeTilePlan()` creates an empty instance.
- `$bounds<-` sets the `n x 4` bounds matrix (`xmin`, `xmax`, `ymin`, `ymax`). Setting bounds triggers reinitialization of `@n`, `@dims`, and `@metadata`.
- `$pad / + / -` add spatial padding as with other plans.

Notes

- `@tile_dims` is intentionally not populated. Operations that require uniform tile dimensions (e.g. `extend = TRUE` in [getTile\(\)](#)) are not supported.
- `nrow()` returns `n`, `ncol()` returns `1`, `length()` returns `n`.
- `plot()` works via the existing spatial extent preview.

Note

`@tile_dims` is intentionally not populated. Any method that requires uniform tile dimensions will not work with `freeTilePlan`.

See Also

Other tile plans: [freeTilePlan\(\)](#), [pixelTilePlan-class](#), [pointTilePlan-class](#), [quadtreePlan\(\)](#), [spatialTilePlan-class](#), [tilePlan\(\)](#), [tilePlan-class](#), [tilework-class](#)

Examples

```
tp <- freeTilePlan()
tp$bounds <- rbind(
  c(0, 50, 0, 50),
  c(50, 100, 0, 50),
  c(0, 50, 50, 100),
  c(50, 100, 50, 100)
)
length(tp)
tp[2]
plot(tp)
```

getBoundedData	<i>Get Data Within Bounds</i>
----------------	-------------------------------

Description

Subset or otherwise make available only the data that is within the provided bounds. Accepted bounds types depends on the data.

A related lower-level function is [getTile\(\)](#) which handles data/tile* interactions and bound info collection. [getTile\(\)](#) calls this generic for its data selection capabilities.

Usage

```
## S4 method for signature 'SpatRaster,numeric'
getBoundedData(x, bound, extend = FALSE, fill = NA)

## S4 method for signature 'SpatRaster,SpatExtent'
getBoundedData(x, bound, extend = FALSE, fill = NA)

## S4 method for signature 'SpatVectorProxy,SpatExtent'
getBoundedData(x, bound)
```

Arguments

x	data
bound	bounds to filter with
extend	logical (default = FALSE) whether to extend tile data to reach expected tile dimensions
fill	numeric. if extend = TRUE, what value to fill with

Bounds types

Depending on the method, different bound types are used. Current expected patterns are:

- numeric of length 4 (xmin, xmax, ymin, ymax)
- SpatExtent

SpatRaster snapping

[getBoundedData\(\)](#) is implemented for [SpatRaster](#), [SpatExtent](#) using [terra::window\(\)](#). This uses terra's default snapping behavior (equivalent to snap = "near" in [terra::crop\(\)](#)), with no way to set another strategy.

For more precise boundary control:

- Use pixel-based indexing via the [SpatRaster](#), numeric method
- Use [pixelTilePlan\(\)](#) for exact pixel-level tiling
- Add padding with tiles + pad_value to ensure spatial context
- Use [terra::crop\(\)](#) directly if you need snap = "out" or snap = "in"

Boundary Inclusivity

Adjacent tiles share exact boundaries. Since {tilework} does not know the format or representation of the underlying data, it does not enforce whether those boundaries are inclusive or exclusive. It is up to the `getBoundedData()` implementation to decide how features on shared edges are handled.

The existing {terra} methods do not implement inclusive/exclusive boundary control because raster pixel snapping and padding make exact boundary behavior largely irrelevant for that format. For point or tabular data, features on a shared boundary may appear in multiple tiles unless the implementation applies its own filtering (e.g. `>=` vs `>` comparisons). The tile's grid position can be passed via `get_params` from `getTile()` to inform which edges are interior.

See Also

`getTile()`

Other tile processing: `getTile()`, `tileApply()`, `tileApply-group`, `tileApply-iterator`, `tileApply-plan`

Examples

```
f <- system.file("ex/elev.tif", package = "terra")
r <- terra::rast(f)

pixel_selection <- getBoundedData(r, c(10, 50, 30, 40))
plot(pixel_selection)

extent_selection <- getBoundedData(r, ext(6, 6.5, 49.7, 50))
plot(extent_selection)
```

getTile

Get Tile

Description

Get specific tile(s) from the data based on a `tile*` object. Indexing via `i` and `j` params is used to select the tile(s) to get, in a manner similar to `[]` extraction of bounds from `tile*` objects.

All params are usable with any method except when otherwise stated.

This generic handles data/tile* object interactions. For simple selection of data based on a set of bounds, see the low-level generic `getBoundedData()`.

The character method only works for filetypes readable by {terra}.

New x data types should automatically work with this generic as long as a `getBoundedData()` method is created for it. A specific `getTile()` method is only needed if additional params/operations should be performed before the tile selection, (for example layer selection with raster data).

Usage

```
## S4 method for signature 'token,tilePlan'
getTile(x, tiles, ...)

## S4 method for signature 'ANY,tilePlan'
getTile(x, tiles, i = NULL, j, pad = NULL, sel_params = list(), ...)

## S4 method for signature 'character,tilePlan'
getTile(x, tiles, prefer = NULL, ext = NULL, ...)

## S4 method for signature 'SpatRaster,tilePlan'
getTile(x, tiles, lyr = NULL, extend = FALSE, fill = NA, ...)

## S4 method for signature 'SpatRaster,pointTilePlan'
getTile(x, tiles, lyr = NULL, extend = FALSE, fill = NA, ...)

## S4 method for signature 'SpatRaster,tileGroup'
getTile(x, tiles, i, j, lyr = NULL, ...)

## S4 method for signature 'ANY,tileGroup'
getTile(x, tiles, i, j, ...)

## S4 method for signature 'ANY,tileIterator'
getTile(x, tiles, advance = TRUE, ...)
```

Arguments

x	data
tiles	tile* object
...	additional params to pass to <code>[</code> tilePlan indexing
i	ANY except tileIterator tile vector index or row index if j is also provided
j	ANY except tileIterator tile col index
pad	(optional) additional padding to apply before tile retrieval. Useful for temporarily increasing padding without affecting tile* object.
sel_params	list of named params to pass to the tile <code>[</code> selection call. Rarely needed; covers internal <code>[</code> options such as <code>expand_grid</code> .
prefer	character filepath only character. Hint for file reading ("raster" or "vector"). If provided, skips automatic type detection.
ext	SpatRaster or raster filepath (character) only numeric or SpatExtent (optional) Set an extent before extracting tiles from raster.
lyr	SpatRaster only if provided, which layers/channels to include
extend	pixelTilePlan only logical (default = FALSE). Whether to extend tile data to reach expected tile dimensions
fill	pixelTilePlan only numeric. if extend = TRUE, what value to fill with
advance	tileIterator only logical (default = TRUE). Whether to advance the iterator.

Value

list of tile data

See Also

[getBoundedData\(\)](#)

Other tile processing: [getBoundedData\(\)](#), [tileApply\(\)](#), [tileApply-group](#), [tileApply-iterator](#), [tileApply-plan](#)

Examples

```
f <- system.file("ex/elev.tif", package = "terra")
r <- terra::rast(f)
tp <- tilePlan("pixel")
tp$pxdims <- dim(r)[1:2]
tp$nrows <- 10
tp$ncols <- 10

# get tiles from specific array grid indices.
tile_list <- getTile(f, tp, i = 3, j = 5)
force(tile_list)
plot(tile_list[[1]])

# get tiles via iteration
iter <- tileIterator(tp, batch_size = 3)
b1 <- getTile(r, iter) # get first batch of 3...
b2 <- getTile(r, iter) # get second batch of 3...
```

intersect

Find tiles intersecting a spatial region

Description

Return a `tileSelection` of tiles whose padded bounds intersect the query region `y`. Tile indices are available via `$tile` on the result.

Accepts any object coercible via `ext()` for an axis-aligned query, or a `SpatVector` polygon for an exact query (e.g. a back-projected parallelogram from a rotated affine crop).

For `SpatVector` input, an AABB pre-cull is applied first, then exact polygon-rectangle intersection is tested on candidates via `terra::relate()`.

`spatialTilePlan` uses an analytic grid formula for the AABB case ($O(1)$ range computation). `freeTilePlan` uses a vectorized bounds check. All other `tilePlan` subclasses fall back to building polygons via `as.polygons()`.

Usage

```
## S4 method for signature 'tilePlan,ANY'
intersect(x, y)

## S4 method for signature 'freeTilePlan,ANY'
intersect(x, y)

## S4 method for signature 'spatialTilePlan,ANY'
intersect(x, y)
```

Arguments

x tilePlan-inheriting object
y spatial query – any object coercible via `ext()`, or a `SpatVector` polygon

Value

`tileSelection`. Use `$tile` to access tile indices.

See Also

Other `tile*` methods: [arith](#), [as.polygons](#), [bracket](#), [centroids](#), [dim](#), [dollar](#), [double_bracket](#), [ext](#), [plot](#)

Examples

```
# spatialTilePlan -- axis-aligned query
tp <- spatialTilePlan(ext = c(0, 100, 0, 100), n = 16)
e <- ext(20, 60, 20, 60)
sel <- intersect(tp, e)
sel$tile      # indices of intersecting tiles
length(sel)   # number of tiles hit
plot(sel)
plot(e, add = TRUE, border = "cyan")

# with padding -- tiles near the border are included
tp_pad <- tp + 5
sel_pad <- intersect(tp_pad, terra::ext(20, 60, 20, 60))
length(sel_pad) >= length(sel) # TRUE: padding pulls in border tiles

# freeTilePlan -- quadtree bounds
fp <- freeTilePlan()
fp$bounds <- rbind(
  c(0, 50, 0, 50),
  c(50, 100, 0, 50),
  c(0, 50, 50, 100),
  c(50, 100, 50, 100)
)
intersect(fp, terra::ext(40, 60, 40, 60))$tile # all 4 tiles touch the centre

# SpatVector polygon query (e.g. back-projected rotated crop)
```

```

corners <- rbind(c(30, 10), c(70, 10), c(90, 50), c(10, 50), c(30, 10))
poly <- terra::vect(corners, type = "polygons")
sel_poly <- intersect(tp, poly)
sel_poly$tile
plot(sel_poly)
plot(poly, add = TRUE, border = "cyan")

```

iterSplit	<i>Create multiple walkers from a single iterator</i>
-----------	---

Description

Utility function to create multiple independent walkers for parallel processing. Each iterator will have the same underlying iterator but independent state.

Usage

```

## S4 method for signature 'tileIterator'
iterSplit(tiles, n, batch_size = NULL, distribute = TRUE, ...)

```

Arguments

tiles	tileIterator object
n	integer. Number of iterators to create
batch_size	integer (optional). Assign a batch size for each iterator. If not provided, inherits same batch_size as source tiles.
distribute	logical (default = TRUE). If TRUE, distribute tiles evenly across iterators. Otherwise return multiple true copies.
...	additional params to pass (none implemented)

See Also

Other tile orchestration: [tileGroup\(\)](#), [tileGroup-class](#), [tileIterator\(\)](#), [tileIterator-class](#)

parallel_params	<i>Parallel Processing Parameters</i>
-----------------	---------------------------------------

Description

[tileApply\(\)](#) is parallelized through either [future.apply::future_lapply\(\)](#) (default) or [BiocParallel::bplapply\(\)](#). Additional parameters to control parallelization can be passed as a list through the `parallel_params` argument in `tileApply()` methods.

Usage

```
getTileworkParMethod()  
  
setTileworkParMethod(method)
```

Arguments

method character. Either "future" (default) or "biocparallel". Which parallelization framework to use.

Value

getTileworkParMethod returns character. The parallelization method being used.
setTileworkParMethod returns NULL invisibly

Parameters

Available parameters for the parallel_params list:

- method - character. Either "future" (default) or "biocparallel". Which parallelization framework to use. Overrides setTileworkParMethod()
- future_seed - logical (default = TRUE). Passed to the future.seed parameter in [future.apply::future_lapply\(\)](#). Only used when method = "future".
- BPPARAM - a {BiocParallel} parallelization parameter object. Only used when method = "biocparallel".

Additional parameters relevant to the underlying functions may also be passed in the list and will be forwarded via

Package Options

- "tilework.warn_sequential" - logical (default = TRUE). Whether to warn when using parallelization defaults that run sequentially (i.e., [future::sequential\(\)](#) or [BiocParallel::SerialParam\(\)](#)).
- "tilework.par_method" - character (default = "future"). Either "future" or "biocparallel". Which parallelization framework to use.
- "tilework.bpparam" - a {BiocParallel} parameter object. Set this to automatically use a specific BPPARAM, analogous to setting a [future::plan\(\)](#) for the {future} backend.

See Also

[tileApply\(\)](#)

Other parallel settings: [tilework_management](#)

Examples

```
getTileworkParMethod()

# Set to BiocParallel
setTileworkParMethod("biocparallel")
getTileworkParMethod()

## Not run:
# Example pseudocode for how to use parallelization

# Using future backend with multisession
future::plan(future::multisession, workers = 4)
tileApply(x, tiles, fun)

# Using BiocParallel backend
setTileworkParMethod("biocparallel")
options(tilework.bpparam = BiocParallel::SnowParam(workers = 4))
tileApply(x, tiles, fun)

# Or override per-call
setTileworkParMethod("future") # global default is future
tileApply(x, tiles, fun,
  parallel_params = list(method = "biocparallel")
) # but use BiocParallel here

# Suppress sequential warnings
options(tilework.warn_sequential = FALSE)

## End(Not run)
```

pixelTilePlan-class *Pixel Tile Plan*

Description

Utility class for defining pixel-exact tiles of images.

Usage

```
pixelTilePlan(pxdims = NULL, ncols = NULL, nrows = NULL, ...)
```

Arguments

pxdims	numeric of length 2. Pixel dimensions c(nrow, ncol) of the image to iterate across. Equivalent to x\$pxdims <- value.
ncols	numeric. Tile width in pixels. Equivalent to x\$ncols <- value.
nrows	numeric. Tile height in pixels. Equivalent to x\$nrows <- value.
...	additional params passed to new().

Slots

pxdims pixel dimensions to iterate across
 n numeric. Number of tiles to create.
 dims numeric. Number of rows/cols in the array of tiles
 tile_dims numeric. Row/col dimensions of each tile
 pad numeric. Tile padding
 metadata data.frame. Metadata per tile

Setup

- pixelTilePlan(pxdims, ncols, nrows) creates an instance. All three params can also be set after construction via \$pxdims<-, \$ncols<-, and \$nrows<-.
- length(), dim(), nrow(), ncol() report tile counts and layout.

Getting tile pixel indices

[i] and [i, j] indexing can be used to select tiles, similarly to a matrix. [] without any indexing will return the entire set of indices as a list. Indices are returned as lists of integer vectors of length 4, with values xmin, xmax, ymin, ymax. Extracted vectors of indices will have metadata attached via attr(). See metadata section below.

padding

+/- can be used to add or subtract padding to each of the tiles. Each bound will be expanded by the pad value. Pad values may only be integer values.

previewing tiles

plot() can be used to check the layout of the tiles.

metadata

The pixelTilePlan object can contain metadata. By default after extent and tiles setup, a column called "tile" will be set up that simply records which tile it is.

- \$ and \$<- can be used to get and set specific metadata, the "pxdims" which are the pixel dimensions of the image to iterate across, "ncols" and "nrows" which are the px dims of a tile, and the "pad" value.
- [[i]] selection will pull specific metadata rows corresponding to the selected tiles.

See Also

Other tile plans: [freeTilePlan\(\)](#), [freeTilePlan-class](#), [pointTilePlan-class](#), [quadtreePlan\(\)](#), [spatialTilePlan-class](#), [tilePlan\(\)](#), [tilePlan-class](#), [tilework-class](#)

Examples

```

x <- pixelTilePlan(pxdims = c(100, 100), ncols = 20)

length(x) # check how many tiles there are
dim(x)
nrow(x)
ncol(x)

# previewing
plot(x)

# tile padding
y <- x + 3
plot(y, alpha = 0.3) # red border shows the image space
# this is now larger than the original space.

# negative padding
z <- x - 5
plot(ext(c(0, 100, -100, 0)))
plot(z, add = TRUE)

# tile selection
x[5]
x[2, 2:3]

# metadata
x$tile
x$fname <- sprintf("tile_%03d.tif", x$tile)
x[[5]]
x[[1:3]]$fname

```

plot

Plot a tilePlan

Description

Plot and preview the tile plan. This is likely to be very slow if there are a lot of tiles (in the neighborhood of >10,000)

Usage

```

## S4 method for signature 'tilePlan,missing'
plot(x, values = "tile", color_as_factor = FALSE, ...)

## S4 method for signature 'freeTilePlan,missing'
plot(x, y, ...)

## S4 method for signature 'pointTilePlan,missing'
plot(x, y, ...)

```

```
## S4 method for signature 'pixelTilePlan,missing'
plot(x, y, ...)

## S4 method for signature 'tileSelection,missing'
plot(x, y, border = "red", ...)
```

Arguments

x	tilePlan to plot
values	character. Metadata item to color tiles as. Default is "tile" which produces the tile index.
color_as_factor	logical (default = FALSE). Whether to convert value to a factor to plot as categorical.
...	additional params to pass
y	not used.
border	tileSelection only color to outline the selected tiles with

See Also

Other tile* methods: [arith](#), [as.polygons](#), [bracket](#), [centroids](#), [dim](#), [dollar](#), [double_bracket](#), [ext](#), [intersect](#)

Examples

```
x <- tilePlan()
ext(x) <- c(0, 100, 0, 100)
length(x) <- 9
plot(x)

x$new_meta <- rev(head(letters, 9))
plot(x, value = "new_meta")
```

pointTilePlan-class *Point Tile Plan*

Description

Tile plan where each tile is centered on a user-supplied (x, y) coordinate with a uniform tile size. The center coordinates are primary — for use cases like survey sampling, object detection patches, or site-centered extractions where the point location is semantically meaningful.

All coordinates, tile dims, and padding live in one consistent reference frame set by \$input ("spatial" for CRS units, "pixel" for pixel indices). The bound type returned by [getTile\(\)](#) is a separate concern controlled by \$output and resolved at extraction time.

Usage

```
pointTilePlan(
  input = c("spatial", "pixel"),
  output = input,
  coords = NULL,
  width = NULL,
  height = NULL,
  ...
)
```

Arguments

input	character. Coordinate space of coords, dims, and padding: "spatial" for CRS units, "pixel" for pixel indices.
output	character. Bound type returned by <code>getTile()</code> . Defaults to input. Cross-mode conversion requires a <code>SpatRaster</code> at <code>getTile()</code> time (or <code>@rast_dims/@extent</code> set on the plan for standalone use).
coords	n x 2 matrix or data frame of tile center coordinates (columns: x, y). Data frames are coerced via <code>as.matrix()</code> . Equivalent to <code>x\$coords <- value</code> after construction.
width	numeric. Uniform tile width in input coordinate units. Equivalent to <code>x\$width <- value</code> after construction.
height	numeric. Uniform tile height in input coordinate units. Equivalent to <code>x\$height <- value</code> after construction.
...	additional params to pass to <code>pointTilePlan</code> class constructor

Slots

coords	matrix. n x 2 matrix of tile center coordinates (columns: x, y).
input	character. Coordinate space of coords, dims, and padding: "spatial" for CRS units, "pixel" for pixel indices.
output	character. Bound type returned by <code>getTile()</code> : "spatial" or "pixel". Defaults to input. Evaluated at <code>getTile()</code> time, not [i].
rast_dims	numeric. <code>c(nrow, ncol)</code> of the reference raster. Required for <code>plot()</code> when <code>input = "pixel"</code> and for cross-mode conversion without a raster.
extent	numeric. <code>c(xmin, xmax, ymin, ymax)</code> of the reference raster. Required for <code>pixel → CRS</code> conversion without a raster.
n	numeric. Number of tiles (equals number of input points).
dims	integer. Always <code>c(n, 1L)</code> .
tile_dims	numeric. Uniform <code>c(height, width)</code> in input coordinate units.
pad	numeric. Tile padding in input coordinate units.
metadata	data.frame. Per-tile metadata; always has "tile", "x", and "y" columns.

Setup

- `pointTilePlan()` creates an instance. `input` defaults to "spatial"; `output` defaults to `input`.
- `$coords<-` sets the $n \times 2$ center coordinate matrix (columns: `x`, `y`). Data frames are also accepted and coerced via `as.matrix()`.
- `$width<-` / `$height<-` set the uniform tile width and height in input coordinate units.
- `$input<-` / `$output<-` change the coordinate / output mode.
- `$nrows<-` / `$ncols<-` are aliases for `$height<-` / `$width<-` for consistency with [pixelTilePlan](#) when using pixel coordinates.
- `$rast_dims<-` sets `c(nrow, ncol)` of the reference raster — required for `plot()` when `input = "pixel"` and for cross-mode `getTile()` calls when no raster is provided.
- `$extent<-` sets `c(xmin, xmax, ymin, ymax)` — required for `pixel` → CRS conversion without a raster.

Coordinate modes and required raster metadata

Scenario	@rast_dims	@extent
<code>input = "spatial", output = "spatial"</code>	not needed	not needed
<code>input = "pixel", output = "pixel"</code>	for <code>plot()</code>	not needed
<code>input = "pixel", output = "spatial"</code>	required	required
<code>input = "spatial", output = "pixel"</code>	required	derived

When [getTile\(\)](#) is called with a `SpatRaster`, the raster's own metadata is used for conversion — `@rast_dims` and `@extent` are only needed for standalone operations (plotting, validation) without a raster.

Getting tile bounds

`[i]` returns bounds in the **input** coordinate space: `SpatExtent` when `input = "spatial"`, `integer[4]` when `input = "pixel"`. `$output` is not consulted at `[i]` time. Use [getTile\(\)](#) for output-mode conversion.

Pixel mode and center ambiguity

When `input = "pixel"`, tile dims should be odd for an unambiguous center pixel. With even dimensions `center ± width/2` produces non-integer bounds that are truncated by `as.integer()`.

metadata

After setting `coords`, metadata is auto-populated with "tile", "x", and "y" columns. Additional columns can be added via `$<-`.

See Also

Other tile plans: [freeTilePlan\(\)](#), [freeTilePlan-class](#), [pixelTilePlan-class](#), [quadtreePlan\(\)](#), [spatialTilePlan-class](#), [tilePlan\(\)](#), [tilePlan-class](#), [tilework-class](#)

Examples

```

tp <- pointTilePlan("spatial")
tp$coords <- cbind(x = c(10, 20, 30), y = c(5, 15, 25))
tp$width <- 5
tp$height <- 5
tp

length(tp)
tp[1]
tp[2:3]
centroids(tp)
plot(tp)

# CRS input, pixel output (raster needed at getTile time)
tp2 <- pointTilePlan(input = "spatial", output = "pixel")
tp2$coords <- cbind(x = c(10, 20), y = c(5, 15))
tp2$width <- 100
tp2$height <- 100

```

quadtreePlan

Adaptive Quadtree Tile Plan

Description

Iteratively subdivides a spatial extent into a freeTilePlan by running FUN on each tile and splitting any tile whose value exceeds threshold. Uses tileApply() internally so each pass can be parallelized via [future::plan\(\)](#).

FUN must return a single numeric scalar per tile — typically a point count, density, or variance measure. Tiles are split into four equal quadrants until all tiles are either below threshold or smaller than min_tile_size.

Usage

```

quadtreePlan(
  x,
  tiles = NULL,
  FUN = nrow,
  threshold,
  min_tile_size = NULL,
  max_depth = 10L,
  verbose = NULL,
  ...
)

```

Arguments

x input data (e.g. SpatVector or file path).

tiles	starting tilePlan. Defines the initial coarse grid.
FUN	function. Applied to each tile; must return a single numeric.
threshold	numeric. Tiles with FUN(tile) > threshold are subdivided.
min_tile_size	numeric. Minimum tile side length (in CRS units). Tiles below this size are kept as leaves regardless of density.
max_depth	integer (default 10L). Maximum subdivision depth.
verbose	logical. Whether to be verbose.
...	additional params passed to tileApply().

Value

A freeTilePlan with an n_records metadata column containing the last FUN value for each leaf tile.

See Also

Other tile plans: [freeTilePlan\(\)](#), [freeTilePlan-class](#), [pixelTilePlan-class](#), [pointTilePlan-class](#), [spatialTilePlan-class](#), [tilePlan\(\)](#), [tilePlan-class](#), [tilework-class](#)

Examples

```
# dummy data
pts <- terra::vect(cbind(x = rnorm(1000, 0, 100), y = rnorm(1000, 0, 100)))
pts <- rbind(pts, terra::shift(pts, dx = 1000, dy = 1000))
plot(pts)
# data must exist on disk
f <- tempfile(fileext = "shp")
terra::writeVector(pts, f)
pts <- terra::vect(f, proxy = TRUE)

fp <- quadtreePlan(pts,
  threshold = 500L,
  min_tile_size = 1
)
plot(fp)
# plot with items per tile
plot(fp, values = "n_records")
```

Description

Utility generic for modifying `tileApply()` calls for streamlining extension with new datatypes and forcing datatype-specific handling to be added in the expected order.

Why

- `tileApply` is a framework function for parallelized computing across tiled data. It allows dispatch on up to 3 signatures (`x`, `y`, and `tile`), allowing flexibility, but at the cost of normally having to deal with combinatorial explosion of methods when extending with a new `x` and `y` datatype. Especially the case when considering compatibility with existing types and specific handling for different tile types.
- Setting specific methods using the following example dispatch chain is ambiguous, with it being unclear whether 1 or 2 is first and whether both will be reached before 3.
 1. (`ANY`, `myclass`, `mytiles`)
 2. (`myclass`, `ANY`, `mytiles`)
 3. (`ANY`, `ANY`, `mytiles`)

`redispatch_tileapply()` solves the above problems by modifying `tileApply()` calls based on the classes of `x` and `y`, appropriate to the provided `tile*` type.

This is done by passing `x` or `y` as input `sig` alongside the `tiles` and dispatching on the combination, and properly routing changes when they are `x` or `y` param specific based on the `param_xy`.

After a `sig` is processed, it is coerced to the `token` class signifying that it is ready for processing and the call is passed back to `tileApply()` which can either proceed with `eval` or pass back to `redispatch_tileapply` if `y` is also provided.

Only when `x` (for single input `tileApply()`) or `x` and `y` are all of class `token`, does the `tileApply()` call fully evaluate.

The default `ANY, ANY` method performs no modifications to the input data or the call.

Usage

```
## S4 method for signature 'SpatRaster,freeTilePlan'
redispatch_tileapply(sig, tiles, ...)

## S4 method for signature 'ANY,ANY'
redispatch_tileapply(
  sig,
  tiles,
  default_get_params = list(),
  param_xy,
  verbose = NULL,
  ...
)

## S4 method for signature 'character,tileGroup'
redispatch_tileapply(sig, tiles, ...)

## S4 method for signature 'SpatVector,tileGroup'
```

```

redispatch_tileapply(sig, tiles, parallel_strategy = c("groups", "tiles"), ...)

## S4 method for signature 'SpatRaster,tileGroup'
redispatch_tileapply(
  sig,
  tiles,
  param_xy,
  parallel_strategy = c("groups", "tiles"),
  ...
)

## S4 method for signature 'ANY,tileGroup'
redispatch_tileapply(
  sig,
  tiles,
  default_get_params = list(),
  default_callback = NULL,
  param_xy,
  parallel_strategy = c("groups", "tiles"),
  verbose = NULL,
  ...
)

## S4 method for signature 'character,tileIterator'
redispatch_tileapply(sig, tiles, ...)

## S4 method for signature 'SpatRaster,tileIterator'
redispatch_tileapply(sig, tiles, param_xy, ...)

## S4 method for signature 'SpatVector,tileIterator'
redispatch_tileapply(sig, tiles, ...)

## S4 method for signature 'ANY,tileIterator'
redispatch_tileapply(
  sig,
  tiles,
  default_get_params = list(),
  default_callback = NULL,
  param_xy,
  verbose = NULL,
  ...
)

## S4 method for signature 'character,tilePlan'
redispatch_tileapply(sig, tiles, ...)

## S4 method for signature 'SpatVector,spatialTilePlan'
redispatch_tileapply(sig, tiles, ...)

```

```
## S4 method for signature 'SpatRaster,spatialTilePlan'
redispatch_tileapply(sig, tiles, ...)

## S4 method for signature 'SpatRaster,pixelTilePlan'
redispatch_tileapply(sig, tiles, ...)

## S4 method for signature 'SpatRaster,pointTilePlan'
redispatch_tileapply(sig, tiles, ...)

## S4 method for signature 'character,tileSelection'
redispatch_tileapply(sig, tiles, ...)

## S4 method for signature 'SpatVector,tileSelection'
redispatch_tileapply(sig, tiles, ...)

## S4 method for signature 'SpatRaster,tileSelection'
redispatch_tileapply(sig, tiles, ...)
```

Arguments

<code>sig</code>	data type to modify the <code>tileApply()</code> call based on.
<code>tiles</code>	<code>tile*</code> object
<code>...</code>	additional params to pass
<code>default_get_params</code>	internal use: list of default <code>getTile()</code> parameters to use with this <code>sig</code> , passed from an upstream (likely more specific) <code>redispatch_tileapply()</code> method. Depending on <code>param_xy</code> , this list of defaults will be merged with <code>get_params_x</code> or <code>get_params_y</code> in ... Duplicate entries in <code>default_get_params</code> will be ignored in favor of <code>get_params_x</code> or <code>get_params_y</code> contents.
<code>param_xy</code>	character. Either "x" or "y" which param it is to modify dispatch on
<code>verbose</code>	verbosity. TRUE, FALSE or "debug" for more info on stack tracing.
<code>parallel_strategy</code>	character. "groups" to parallelize across groups, or "tiles" to parallelize within groups
<code>default_callback</code>	internal use: Default callback function passed from an upstream (likely more specific) <code>redispatch_tileapply()</code> method to use as <code>callback_x</code> or <code>callback_y</code> if none has been provided.

Logic flow between `tileApply()` and `redispatch_tileapply()`

x only case:

1. `tileApply(ANY, missing, ANY, ...)`
2. `redispatch_tileapply(mydata, mytiles, param_xy = "x", ...)`

3. `redispatch_tileapply(ANY, ANY, param_xy = "x", ...)`
4. `tileApply(token, missing, ANY, ...)`
5. evaluation

x and y case:

1. `tileApply(ANY, ANY, ANY)`
2. `redispatch_tileapply(mydata1, mytiles, param_xy = "x")`
3. `redispatch_tileapply(ANY, ANY, param_xy = "x", ...)`
4. `tileApply(token, ANY, ANY)`
5. `redispatch_tileapply(mydata2, mytiles, param_xy = "y")`
6. `redispatch_tileapply(ANY, ANY, param_xy = "y", ...)`
7. `tileApply(token, token, plan)`
8. evaluation

This allows:

- Data type-specific preprocessing, checks, and additional params to be injected into the `tileApply()` call without having to interact with the core `tileApply()` logic for tilewise data selection and parallelization
- Avoid duplicating data type-specific handling code for each of the following `tileApply()` signature types:
 - `(mydata, missing, tiles)`
 - `(mydata, ANY, tiles)`
 - `(token, mydata, tiles)`

Instead, extending developers only have to provide code for a single `redispatch_tileapply(myclass, tiles)` signature, and `tileApply() <-> redispatch_tileapply()` calls will iteratively apply the appropriate preprocessing and x or y routed params.

The x and y routing and token coercion are also handled by the `redispatch_tileapply(ANY, ANY)` method.

See Also

Other tilework extension: [extending_tilework](#), [token-class](#)

spatialTilePlan-class *Spatial Tile Plan*

Description

Utility class that simplifies the setup of tiles across a spatial extent. Tiles are stored in a lightweight format safe to be passed to child processes. Tile SpatExtent objects can be extracted on-demand.

Usage

```
spatialTilePlan(ext = NULL, n = NULL, ...)
```

Arguments

ext	numeric or SpatExtent. Spatial extent to tile across. Equivalent to calling <code>ext(x) <- value</code> after construction.
n	numeric. Desired number of tiles. Equivalent to calling <code>length(x) <- value</code> after construction.
...	additional params passed to <code>new()</code> .

Slots

extent numeric. Spatial extent to tile across.
n numeric. Number of tiles to create.
dims numeric. Number of rows/cols in the array of tiles
tile_dims numeric. Row/col dimensions of each tile
pad numeric. Tile padding
metadata data.frame. Metadata per tile

setup and basic characteristics

A spatialTilePlan needs both a spatial extent to tile across and also a request for a certain number of tiles.

- `spatialTilePlan(ext, n)` is used to create a `spatialTilePlan` instance. `ext` and `n` can also be supplied after construction via `ext()<-` and `length()<-`.
- `ext()` is used to check extent.
- `length()` can be used to find out how many tiles there are.
- `dim()/nrow()/ncol()` basic generics are implemented and return information about how the tiles are arranged.

Note that the number of requested tiles may not be the actual length, since a grid pattern must be followed. However, the number of generated tiles will be AT LEAST the number that is requested. Generated tiles will have as square a shape as possible.

Getting tile extent

[i] and [i, j] indexing can be used to select tiles, similarly to a matrix. [] without any indexing will return the entire set of extents as a list. Extracted extents will have metadata attached via attr(). See metadata section below.

padding

+/- can be used to add or subtract padding to each of the tiles. Note that this value does not affect the setting or retrieval of extent info via ext() and ext()<-. Each bound will be expanded by the padding value. To avoid having to use terra::extend() when padded tiles exceed raster extent, you can decrease the extent by the same size as the padding.

previewing tiles

plot() can be used to check the layout of the tiles.

metadata

The spatialTilePlan object can contain metadata. By default after extent and tiles setup, a column called "tile" will be set up that simply records which tile it is.

- \$ can be used to view a specific type of metadata and the padding value
- \$<- can be used to set additional metadata items and the padding value
- [[i]] selection will pull specific metadata rows corresponding to the selected tiles.

See Also

Other tile plans: [freeTilePlan\(\)](#), [freeTilePlan-class](#), [pixelTilePlan-class](#), [pointTilePlan-class](#), [quadtreePlan\(\)](#), [tilePlan\(\)](#), [tilePlan-class](#), [tilework-class](#)

Examples

```
x <- spatialTilePlan(ext = c(0, 100, 0, 100), n = 8)
force(x)

length(x) # how many were actually generated? AT LEAST n
dim(x)
nrow(x)
ncol(x)

# previewing
plot(x)

# tile padding
y <- x + 10
plot(ext(x), border = "red")
plot(y, alpha = 0.3, add = TRUE)
# this is now larger than the original space
ext(y) <- ext(y) - 10
plot(y, alpha = 0.3)
plot(ext(x), add = TRUE, border = "red")
```

```

# now this does not exceed the image

# negative padding
z <- x - 5
plot(ext(x), border = "red")
plot(z, add = TRUE)

# tile selection
x[5]
x[1, 2:3]

# metadata
x$tile
x$fname <- sprintf("tile_%03d.tif", x$tile)
x[[5]]
x[[1:3]]$fname

# selected tiles carry metadata as attributes
attr(x[4][[1]], "tile")
attr(x[4][[1]], "fname")

```

tileApply-group

Hierarchical Tile Group Processing

Description

Apply functions across tileGroup objects with control over parallelization strategy. Useful when tiles are organized into logical groups that need different processing or aggregation.

token is a stand-in for any input data class (e.g. SpatRaster, SpatExtent, filepath, etc). See [redispatch_tileapply](#) and [extending_tilework](#) for further information.

Usage

```

## S4 method for signature 'token,missing,tileGroup'
tileApply(
  x,
  tiles,
  FUN,
  get_params_x = list(),
  parallel_strategy = c("groups", "tiles"),
  setup_FUN = NULL,
  group_FUN = NULL,
  callback_x = NULL,
  log = FALSE,
  logpath = getTileworkLogDir(),
  simplify = FALSE,
  parallel_params = list(),
  verbose = NULL,

```

```

    ...
)

## S4 method for signature 'token,token,tileGroup'
tileApply(
  x,
  y,
  tiles,
  FUN,
  get_params_x = list(),
  get_params_y = list(),
  pad_y = NULL,
  parallel_strategy = c("groups", "tiles"),
  setup_FUN = NULL,
  group_FUN = NULL,
  callback_x = NULL,
  callback_y = NULL,
  log = FALSE,
  logpath = getTileworkLogDir(),
  simplify = FALSE,
  parallel_params = list(),
  verbose = NULL,
  ...
)

```

Arguments

x	input data l
tiles	tileGroup object
FUN	function to apply to each tile
get_params_x	named list. Additional params to pass to getTile() for x
parallel_strategy	character. "groups" to parallelize across groups, or "tiles" to parallelize within groups
setup_FUN	function. Optional per-group initialization function when parallel_strategy = "groups". Output is accessible within FUN as .SETUP_OUT
group_FUN	function. Optional function to apply to each group's results
callback_x	function. Optional preprocessing function for x per worker when parallel_strategy = "groups"
log	logical. Whether to log processing steps
logpath	character. Log file path (if log = TRUE)
simplify	logical. Whether to flatten group results into single list. Group names will not be retained.
parallel_params	named param list. See parallel_params
verbose	verbosity. TRUE, FALSE or "debug" for more info on stack tracing.

...	additional params to pass to [
y	input data 2 (optional)
get_params_y	named list. Additional params to pass to getTile() for y
pad_y	numeric. Additional padding applied to y tiling so x has full spatial context of y
callback_y	function. Optional preprocessing function for y per worker when parallel_strategy = "groups"

Parallelization Strategies

- "groups" - Process groups in parallel, tiles within groups sequentially
- "tiles" - Process groups sequentially, tiles within groups in parallel

Special Function Parameters

Your FUN can optionally include these special parameters:

- .I - tile number (integer)
- .TILE - tile bounds/metadata
- .R - tile row number
- .C - tile column number
- .GROUP - current group name (character)
- .SETUP_OUT - the output of setup_FUN

Your setup_FUN can optionally include these special parameters:

- .GROUP - current group name (character)
- .X - the input x object
- .Y - the input y object (when provided)

See Also

[tileApply](#), [tileGroup\(\)](#), [tileGroup](#)

Other tile processing: [getBoundedData\(\)](#), [getTile\(\)](#), [tileApply\(\)](#), [tileApply-iterator](#), [tileApply-plan](#)

Examples

```
f <- system.file("ex/elev.tif", package = "terra")
r <- terra::rast(f)

# Create tile plan
tp <- tilePlan("spatial")
ext(tp) <- ext(r)
length(tp) <- 16

# Organize into groups (e.g., by geographic region)
tg <- tileGroup(tp, groups = list(
  "north" = 1:8, # northern tiles
```

```

    "south" = 9:16, # southern tiles
    "corners" = c(1, 4, 13, 16) # corner tiles
  ))

# Process groups in parallel, with group-level aggregation
results <- tileApply(r,
  tiles = tg,
  parallel_strategy = "groups",
  FUN = function(tile, .I, .GROUP) {
    # Process individual tile
    list(
      tile_id = .I,
      group = .GROUP,
      stats = terra::global(tile, c("mean", "sd"), na.rm = TRUE)
    )
  },
  group_FUN = function(group_results, .GROUP) {
    # Aggregate results within each group
    means <- sapply(group_results, function(x) x$stats$mean)
    list(
      group = .GROUP,
      n_tiles = length(group_results),
      group_mean = mean(means),
      group_range = range(means)
    )
  }
)

# Results organized by group
str(results)

```

tileApply-iterator	<i>Streaming Tile Processing with Iterators</i>
--------------------	---

Description

Apply functions using `tileIterator` objects for memory-constrained batch processing. Ideal for very large datasets or when you need fine control over processing workflow.

`token` is a stand-in for any input data class (e.g. `SpatRaster`, `SpatExtent`, `filepath`, etc). See [redispatch_tileapply](#) and [extending_tilework](#) for further information.

Usage

```

## S4 method for signature 'token,missing,tileIterator'
tileApply(
  x,
  tiles,
  FUN,
  get_params_x = list(),

```

```

    setup_FUN = NULL,
    callback_x = NULL,
    log = FALSE,
    logpath = getTileworkLogDir(),
    simplify = FALSE,
    parallel_params = list(),
    verbose = NULL,
    ...
)

## S4 method for signature 'token,token,tileIterator'
tileApply(
  x,
  y,
  tiles,
  FUN,
  get_params_x = list(),
  get_params_y = list(),
  pad_y = NULL,
  setup_FUN = NULL,
  callback_x = NULL,
  callback_y = NULL,
  log = FALSE,
  logpath = getTileworkLogDir(),
  simplify = FALSE,
  parallel_params = list(),
  verbose = NULL,
  ...
)

```

Arguments

x	input data l
tiles	tileIterator object
FUN	function to apply to each batch of tiles
get_params_x	named list. Additional params to pass to getTile() for x
setup_FUN	function. Optional per-worker initialization function. Output is accessible within FUN as .SETUP_OUT
callback_x	function, (advanced). If provided, programmatic escape hatch for preprocessing x per worker before setup_FUN and batch streamed processing of FUN begins.
log	logical. Whether to log processing steps
logpath	character. Log file path (if log = TRUE)
simplify	logical. Whether to flatten results
parallel_params	named param list. See parallel_params
verbose	verbosity. TRUE, FALSE or "debug" for more info on stack tracing.

...	additional params to pass to [
y	input data 2 (optional)
get_params_y	named list. Additional params to pass to getTile() for y
pad_y	numeric. Additional padding applied to y tiling so x has full spatial context of y
callback_y	function, (advanced). If provided, programmatic escape hatch for preprocessing y per worker before setup_FUN and batch streamed processing of FUN begins.

Worker Distribution

[iterSplit\(\)](#) is run with `n = future::nbrOfWorkers()` to distribute the tiles to process across the expected number of workers, with each worker processing its assigned tiles in batches.

Special Function Parameters

Your FUN can optionally include these special parameters:

- .I - tile number (integer)
- .TILE - tile bounds/metadata
- .R - tile row number
- .C - tile column number
- .POSITION - batch position range (start, end)
- .BATCH - batch number within worker
- .SETUP_OUT - the output of setup_FUN

Your setup_FUN can optionally include these special parameters:

- .W - worker number
- .X - the input x object.
- .Y - the input y object (when provided)

See Also

[tileApply](#), [tileIterator\(\)](#), [tileIterator](#)

Other tile processing: [getBoundedData\(\)](#), [getTile\(\)](#), [tileApply\(\)](#), [tileApply-group](#), [tileApply-plan](#)

Examples

```
f <- system.file("ex/elev.tif", package = "terra")
r <- terra::rast(f)

# Create pixel tile plan
tp <- tilePlan("pixel")
tp$pxdims <- dim(r)[1:2]
tp$nrows <- 30
tp$ncols <- 30

# Create iterator for batch processing
```

```

iter <- tileIterator(tp, batch_size = 5)

# Process with worker initialization
results <- tileApply(r,
  tiles = iter,
  setup_FUN = function(.W, .X) {
    # Initialize per-worker state
    list(
      worker_id = .W,
      start_time = Sys.time(),
      raster_info = list(nrow = nrow(.X), ncol = ncol(.X))
    )
  },
  FUN = function(batch, .BATCH, .POSITION, .SETUP_OUT) {
    # Process batch of tiles
    batch_stats <- lapply(batch, function(tile) {
      terra::global(tile, "mean", na.rm = TRUE)[[1]]
    })

    list(
      worker = .SETUP_OUT$worker_id,
      batch_num = .BATCH,
      tiles_processed = .POSITION,
      batch_mean = mean(unlist(batch_stats))
    )
  }
)

# Check results structure
str(results)

# Example: Streaming processing for memory management
large_iter <- tileIterator(tp, batch_size = 3)
processed_count <- 0

while (large_iter$has_next) {
  batch <- getTile(r, large_iter)

  # Process batch
  batch_results <- lapply(batch, function(tile) {
    # Your processing here
    terra::global(tile, "mean")
  })

  processed_count <- processed_count + length(batch)
  cat("Processed", processed_count, "of", length(tp), "tiles\n")
}

```

Description

Apply functions across tilePlan-inheriting objects.

token is a stand-in for any input data class (e.g. SpatRaster, SpatExtent, filepath, etc). See [redispatch_tileapply](#) and [extending_tilework](#) for further information.

Usage

```
## S4 method for signature 'token,missing,tilePlan'
tileApply(
  x,
  tiles,
  FUN,
  get_params_x = list(),
  log = FALSE,
  logpath = getTileworkLogDir(),
  parallel_params = list(),
  verbose = NULL,
  ...
)

## S4 method for signature 'token,token,tilePlan'
tileApply(
  x,
  y,
  tiles,
  FUN,
  get_params_x = list(),
  get_params_y = list(),
  pad_y = NULL,
  log = FALSE,
  logpath = getTileworkLogDir(),
  parallel_params = list(),
  verbose = NULL,
  ...
)

## S4 method for signature 'token,missing,tileSelection'
tileApply(
  x,
  tiles,
  FUN,
  get_params_x = list(),
  log = FALSE,
  logpath = getTileworkLogDir(),
  parallel_params = list(),
  verbose = NULL,
  ...
)
```

```
## S4 method for signature 'token,token,tileSelection'
tileApply(
  x,
  y,
  tiles,
  FUN,
  get_params_x = list(),
  get_params_y = list(),
  pad_y = NULL,
  log = FALSE,
  logpath = getTileworkLogDir(),
  parallel_params = list(),
  verbose = NULL,
  ...
)
```

Arguments

x	input data 1
tiles	tilePlan inheriting object (spatialTilePlan or pixelTilePlan)
FUN	function to apply to each tile
get_params_x	named list. Additional params to pass to getTile() for x
log	logical. Whether to log processing steps
logpath	character. Log file path (if log = TRUE)
parallel_params	named param list. See parallel_params
verbose	verbosity. TRUE, FALSE or "debug" for more info on stack tracing.
...	additional params to pass to [
y	input data 2 (optional)
get_params_y	named list. Additional params to pass to getTile() for y
pad_y	numeric. Additional padding applied to y tiling so x has full spatial context of y

Special Function Parameters

Your FUN can optionally include these special parameters:

- .I - tile number (integer)
- .TILE - tile bounds/metadata
- .R - tile row number
- .C - tile column number

See Also

[tileApply](#), [tilePlan\(\)](#), [spatialTilePlan](#), [pixelTilePlan](#)

Other tile processing: [getBoundedData\(\)](#), [getTile\(\)](#), [tileApply\(\)](#), [tileApply-group](#), [tileApply-iterator](#)

Examples

```
f <- system.file("ex/elev.tif", package = "terra")
r <- terra::rast(f)

# Spatial tiling example
tp_spatial <- tilePlan("spatial")
ext(tp_spatial) <- ext(r)
length(tp_spatial) <- 4

# Apply function with tile metadata
results <- tileApply(r, tiles = tp_spatial, FUN = function(tile, .I, .R, .C) {
  list(
    tile_id = .I,
    position = paste0("row_", .R, "_col_", .C),
    mean_value = terra::global(tile, "mean", na.rm = TRUE)[[1]]
  )
})
force(results)

# Pixel tiling example
tp_pixel <- tilePlan("pixel")
tp_pixel$pxdims <- dim(r)[1:2]
tp_pixel$nrows <- 50
tp_pixel$ncols <- 50

# Save tiles to disk
outdir <- file.path(tempdir(), "tiles")
dir.create(outdir, showWarnings = FALSE)

tileApply(r, tiles = tp_pixel, FUN = function(tile, .I) {
  filename <- file.path(outdir, sprintf("tile_%03d.tif", .I))
  terra::writeRaster(tile, filename, overwrite = TRUE)
  return(filename)
})

list.files(outdir)
unlink(outdir, recursive = TRUE)
```

tileGroup

Create a Tile Group

Description

Organize tiles from a tilePlan into hierarchical groups for batch processing. Groups can represent spatial regions, processing stages, or any logical organization of tiles.

Usage

```
tileGroup(tp, groups = list(), metadata = data.frame())
```

Arguments

tp	A tilePlan object
groups	Named list where each element contains tile indices for that group
metadata	Optional data.frame with group metadata

See Also

Other tile orchestration: [iterSplit\(\)](#), [tileGroup-class](#), [tileIterator\(\)](#), [tileIterator-class](#)

Examples

```
tp <- tilePlan("spatial")
ext(tp) <- c(0, 100, 0, 100)
length(tp) <- 16

# Create groups
tg <- tileGroup(tp, groups = list(
  "g1" = 1:4, # vector indexing
  "g2" = list(2, 1:4), # ij indexing
  "g3" = list(2:4, 1:2) # selection overlaps are allowed
  # (not all tiles need to be selected)
))

# length() returns number of groups
length(tg)

# Get group bounds:
tg[, "g1"] # first 4
tg[, "g2"] # next 4
tg[, "g3"] # specific 6
# not recommended for large groups

tg[c(3, 1), "g3"] # get nth item in group

# Set active group for shorthand indexing
tg$active <- "g1"

# length() is based on group length when active is set
length(tg)

tg[2] # Position 2 from g1

# iterator can be created when active group is set
iter <- tileIterator(tg, batch_size = 2)
iter$next_batch()
iter$next_batch()
iter$next_batch() # no more items

tg$active <- NULL # Clear active group by setting NULL
```

tileGroup-class	<i>Tile Group</i>
-----------------	-------------------

Description

Class for organizing tiles into hierarchical groups for batch processing. Groups can represent spatial regions, processing stages, or any logical organization of tiles.

Slots

`tp` tilePlan. The underlying tilePlan-inheriting object
`groups` list. Named list where each element contains tile indices for that group
`active` character. Name of a group to set as active for `[i]` shorthand indexing and `length()`.
`metadata` data.frame. Metadata about each group

See Also

Other tile orchestration: [iterSplit\(\)](#), [tileGroup\(\)](#), [tileIterator\(\)](#), [tileIterator-class](#)

tileIterator	<i>Stateful Tile Iterator</i>
--------------	-------------------------------

Description

Create a stateful iterator that maintains an internal position and yields batches of tiles on demand. Uses closures for serializable stateful behavior without external dependencies. Useful for streaming processing, memory-constrained environments, or when you want to process tiles in smaller chunks.

This is a structure that works with any underlying tilePlan. tileGroup is also accepted when an \$active group has already been set, or a group is specified via tileIterator(active_group)

Note that tileIterator contains its own state and an independent copy of it cannot be generated by assigning to a new variable. Instead, use either the \$copy() method or iterSplit().

Usage

```
## S4 method for signature 'tileIterator'
x$name

## S4 replacement method for signature 'tileIterator'
x$name <- value

tileIterator(
  tiles = NULL,
  position = 0,
  bound = NULL,
```

```

    batch_size = 1,
    active_group = NULL
  )

```

Arguments

x	tileIterator
name	Name of method or attribute to access (See Methods section)
value	value to set for an attribute (See Methods section)
tiles	A tile* object
position	numeric. Set the starting position of the iterator. Default is 0.
bound	numeric, length of 2. Min and max indexing bounds through which to iterate. Default is c(1, length(tiles))
batch_size	Integer. Number of tiles to return per call to next_batch()
active_group	character (optional). Only used when tiles is a tileGroup. Syntactic sugar for setting the active group of the tileGroup before using it.

Value

A tileIterator function with attached methods

Methods

- `[]/[]<-` or `$tiles/$tiles<-`
 - get or set a tilePlan or tileGroup (with \$active set). `[]` notation is just shorthand.
- `$position/$position<-`
 - get and set current position of iterator
- `$batch_size/$batch_size<-`
 - get and set batch size.
- `$bound/$bound<-`
 - get and set iteration tile index bounds. This is a numeric vector of 2 (min, max)
- `$next_batch()/ $next_indices()`
 - get next batch or just their tile indices.
 - * Has an advance param (default = TRUE) that advances iterator position.
 - * When there are no more tiles, gracefully returns empty list
- `$peek_batch()/ $peek_indices`
 - Get next batch or indices without advancing position. When there are no more tiles, gracefully returns empty integer
- `$has_next`
 - Return logical if there are still tiles left to iterate through (if `position < max(bound)`). Useful for while looping.
- `$total_tiles`
 - Return total number of tiles in bound

- `$remaining`
 - Return number of remaining indices to traverse.
- `$progress`
 - Return current percentage of position traversal across bound
- `$reset()`
 - Reset position to 1 before the lower bound
- `$copy()`
 - Create a new `tileIterator` with the same settings (deep copy)

Usage Patterns

- Stream processing large datasets
- Memory-constrained batch processing
- Progressive/incremental analysis
- Checkpoint and resume workflows
- Parallel processing with `foreach`

See Also

[tileIterator](#)

Other tile orchestration: [iterSplit\(\)](#), [tileGroup\(\)](#), [tileGroup-class](#), [tileIterator-class](#)

Examples

```
# Create a spatial tile iterator
tp <- tilePlan("spatial")
ext(tp) <- c(0, 100, 0, 100)
length(tp) <- 16

# Create a iterator that processes 3 tiles at a time
iter <- tileIterator(tp, batch_size = 3)

# Check status
iter
iter$has_next
iter$remaining
iter$progress

# Get next batch
batch1 <- iter$next_batch()
length(batch1) # 3 tiles

# Check updated status
iter$progress # position has advanced

# Peek at next batch without advancing
peek <- iter$peek_batch()
peek
```

```

iter$position # unchanged

# serialize and unserialize, preserving state
temp <- tempfile()
saveRDS(iter, temp)
rm(iter)
iter <- readRDS(temp)

# Next batch of indices
iter$next_indices()

# Peek next batch of indices
iter$peek_indices()

# Process all remaining batches
while (iter$has_next) {
  batch <- iter$next_batch()
  cat("Processing batch of", length(batch), "tiles\n")
}

# Reset and try different batch size
iter$reset()
iter$batch_size <- 5

# Apply function across all batches
while (iter$has_next) {
  batch <- iter$next_batch()
  cat("Processing batch of", length(batch), "tiles\n")
}

# tileIterator with tileGroup
tg <- tileGroup(tp, groups = list(
  "g1" = c(2, 4, 6, 8, 10, 12, 14, 16),
  "g2" = 1:16
))

tg$active <- "g1"
iter <- tileIterator(tg, batch_size = 3)
iter
iter$next_batch()

# iterator splitting
siter <- iterSplit(iter, n = 2)
siter

```

tileIterator-class	<i>tileIterator</i>
--------------------	---------------------

Description

A stateful iterator that progresses through tiles of an underlying tilePlan (or tileGroup if \$active is set) object upon every call to \$next_batch()

The closures that power this functionality are stored in @funs. The stateful position handling is also internalized within.

Slots

funs list of closure methods

See Also

[tileIterator](#)

Other tile orchestration: [iterSplit\(\)](#), [tileGroup\(\)](#), [tileGroup-class](#), [tileIterator\(\)](#)

tilePlan	Create a Tiling Plan
----------	----------------------

Description

Create a Tiling Plan

Usage

```
tilePlan(type = c("spatial", "pixel", "point", "free"), ...)
```

Arguments

type	character. One of "spatial", "pixel", "point", "free". Type of plan to create.
...	additional params passed to the specific constructor: spatialTilePlan() , pixelTilePlan() , pointTilePlan() , or freeTilePlan() .

See Also

[spatialTilePlan](#), [pixelTilePlan](#), [pointTilePlan](#), [freeTilePlan](#)

Other tile plans: [freeTilePlan\(\)](#), [freeTilePlan-class](#), [pixelTilePlan-class](#), [pointTilePlan-class](#), [quadtreePlan\(\)](#), [spatialTilePlan-class](#), [tilePlan-class](#), [tilework-class](#)

Examples

```
tilePlan("spatial")
tilePlan("pixel")
tilePlan("point")
tilePlan("free")
```

tilePlan-class	<i>Tile Plan</i>
----------------	------------------

Description

Virtual parent class for tile planning objects. These objects are for planning tiles/patches of data to be operated over. Objects are indexable across rows/columns of tiles and as a vector of tiles (similarly to a matrix).

[indexing returns a set of bounds information for selection of the data. Tiles and bounds information are generated in a lazy fashion based on plan parameters, avoiding overhead with large amounts of tiles.

Slots

n numeric. Number of tiles to create.
 dims numeric. Number of rows/cols in the array of tiles
 tile_dims numeric. Row/col dimensions of each tile
 pad numeric. Tile padding
 metadata data.frame. Metadata per tile

See Also

[spatialTilePlan](#) and [pixelTilePlan](#) for concrete classes dealing with spatial and pixel-exact tiling respectively. [tilePlan\(\)](#) for creation of these objects.

Other tile plans: [freeTilePlan\(\)](#), [freeTilePlan-class](#), [pixelTilePlan-class](#), [pointTilePlan-class](#), [quadtreesPlan\(\)](#), [spatialTilePlan-class](#), [tilePlan\(\)](#), [tilework-class](#)

tilework-class	<i>Virtual Class</i> tilework
----------------	-------------------------------

Description

The tilework class is a class contained by all tile* classes in the **tilework** package. It is a "virtual" class.

See Also

Other tile plans: [freeTilePlan\(\)](#), [freeTilePlan-class](#), [pixelTilePlan-class](#), [pointTilePlan-class](#), [quadtreesPlan\(\)](#), [spatialTilePlan-class](#), [tilePlan\(\)](#), [tilePlan-class](#)

tilework_management	<i>tilework management</i>
---------------------	----------------------------

Description

A set of utilities to control the location and naming of automated tilework outputs. These are mostly to do with logging information right now.

Usage

```
getTileworkLogDir()

setTileworkLogDir(path)

getTileworkJobID(advance = FALSE)
```

Arguments

- path directory path to use
- advance logical (default = FALSE). Whether to consume this job ID

Value

getTileworkLogDir the logging directory to use
setTileworkLogDir returns the previous logging directory invisibly
getTileworkJobID returns the next job ID to use. If advance = FALSE this call will be treated as a peek and will not consume the ID.

See Also

Other parallel settings: [parallel_params](#)

token-class	token <i>Class</i>
-------------	--------------------

Description

Utility class for flagging a piece of data as being ready for processing. This is internal machinery that is mainly useful for forcing S4 dispatch to progress in the expected order and should not be interacted with by end users.

This class and related methods are exported so developers are able to write extending methods for tileApply() where this utility class is used.

Usage

```
## S4 method for signature 'token,missing,missing,missing'  
x[i, j, ..., drop = TRUE]
```

Arguments

x, i, j, ..., drop not used

Slots

data ANY. The wrapped data object.

See Also

Other tilework extension: [extending_tilework](#), [redispatch_tileapply](#), [SpatRaster](#), [freeTilePlan-method](#)

Examples

```
# Flag as being ready for processing  
x <- as(letters, "token")  
  
# Retrieve wrapped data  
x[]
```

Index

- * **parallel settings**
 - parallel_params, 22
 - tilework_management, 55
- * **tile orchestration**
 - iterSplit, 22
 - tileGroup, 47
 - tileGroup-class, 49
 - tileIterator, 49
 - tileIterator-class, 52
- * **tile plans**
 - freeTilePlan, 15
 - freeTilePlan-class, 15
 - pixelTilePlan-class, 24
 - pointTilePlan-class, 27
 - quadtreePlan, 30
 - spatialTilePlan-class, 36
 - tilePlan, 53
 - tilePlan-class, 54
 - tilework-class, 54
- * **tile processing**
 - getBoundedData, 17
 - getTile, 18
 - tileApply-group, 38
 - tileApply-iterator, 41
 - tileApply-plan, 44
- * **tile* methods**
 - arith, 3
 - as.polygons, 4
 - bracket, 5
 - centroids, 8
 - dim, 9
 - dollar, 10
 - double_bracket, 12
 - ext, 13
 - intersect, 20
 - plot, 26
- * **tilework extension**
 - extending_tilework, 14
 - redispatch_tileapply, SpatRaster, freeTilePlan, tileMethod, 31
 - token-class, 55
 - +, pixelTilePlan, numeric-method (arith), 3
 - +, pointTilePlan, numeric-method (arith), 3
 - +, tileGroup, numeric-method (arith), 3
 - +, tilePlan, numeric-method (arith), 3
 - +, tileSelection, numeric-method (arith), 3
 - , tileGroup, numeric-method (arith), 3
 - , tilePlan, numeric-method (arith), 3
 - , tileSelection, numeric-method (arith), 3
 - [, 19, 40, 43, 46
 - [(bracket), 5
 - [, freeTilePlan, numeric, numeric, missing-method (bracket), 5
 - [, pixelTilePlan, numeric, numeric, missing-method (bracket), 5
 - [, pointTilePlan, numeric, missing, missing-method (bracket), 5
 - [, spatialTilePlan, numeric, numeric, missing-method (bracket), 5
 - [, tileGroup, .index, missing, missing-method (bracket), 5
 - [, tileGroup, missing, .index, missing-method (bracket), 5
 - [, tileGroup, missing, missing, missing-method (bracket), 5
 - [, tileGroup, numeric, .index, missing-method (bracket), 5
 - [, tileIterator, missing, missing, missing-method (bracket), 5
 - [, tilePlan, missing, missing, logical-method (bracket), 5
 - [, tilePlan, missing, missing, missing-method (bracket), 5
 - [, tilePlan, missing, missing, numeric, logical-method (bracket), 5

- (bracket), 5
- [,tilePlan,missing,numeric,missing-method (bracket), 5
- [,tilePlan,numeric,missing,logical-method (bracket), 5
- [,tilePlan,numeric,missing,missing-method (bracket), 5
- [,tilePlan,numeric,numeric,logical-method (bracket), 5
- [,tilePlan,numeric,numeric,missing-method (bracket), 5
- [,tileSelection,numeric,missing,logical-method (bracket), 5
- [,tileSelection,numeric,missing,missing-method (bracket), 5
- [,token,missing,missing,missing-method (token-class), 55
- [<-,tileIterator,missing,missing-method (bracket), 5
- [[(double_bracket), 12
- [[,tilePlan,missing,character-method (double_bracket), 12
- [[,tilePlan,missing,missing-method (double_bracket), 12
- [[,tilePlan,numeric,character-method (double_bracket), 12
- [[,tilePlan,numeric,missing-method (double_bracket), 12
- [[<-,tilePlan,missing,character-method (double_bracket), 12
- [[<-,tilePlan,numeric,character-method (double_bracket), 12
- \$,freeTilePlan-method (dollar), 10
- \$,pixelTilePlan-method (dollar), 10
- \$,pointTilePlan-method (dollar), 10
- \$,tileGroup-method (dollar), 10
- \$,tileIterator-method (tileIterator), 49
- \$,tilePlan-method (dollar), 10
- \$,tileSelection-method (dollar), 10
- \$<-,freeTilePlan-method (dollar), 10
- \$<-,pixelTilePlan-method (dollar), 10
- \$<-,pointTilePlan-method (dollar), 10
- \$<-,tileGroup-method (dollar), 10
- \$<-,tileIterator-method (tileIterator), 49
- \$<-,tilePlan-method (dollar), 10
- \$<-,tileSelection-method (dollar), 10
- \$pad, 3
- arith, 3, 5, 7, 9, 10, 12–14, 21, 27
- as.polygons, 4, 4, 7, 9, 10, 12–14, 21, 27
- as.polygons,freeTilePlan-method (as.polygons), 4
- as.polygons,pointTilePlan-method (as.polygons), 4
- as.polygons,spatialTilePlan-method (as.polygons), 4
- as.polygons,tilePlan-method (as.polygons), 4
- batch iteration, 14
- BiocParallel::bplapply(), 22
- bracket, 4, 5, 5, 9, 10, 12–14, 21, 27
- c, 8
- c,tileSelection-method (c), 8
- centroids, 4, 5, 7, 8, 10, 12–14, 21, 27
- centroids,freeTilePlan-method (centroids), 8
- centroids,pixelTilePlan-method (centroids), 8
- centroids,pointTilePlan-method (centroids), 8
- centroids,spatialTilePlan-method (centroids), 8
- centroids,tilePlan-method (centroids), 8
- dim, 4, 5, 7, 9, 9, 12–14, 21, 27
- dim,tilePlan-method (dim), 9
- dollar, 4, 5, 7, 9, 10, 10, 13, 14, 21, 27
- double_bracket, 4, 5, 7, 9, 10, 12, 12, 14, 21, 27
- ext, 4, 5, 7, 9, 10, 12, 13, 13, 21, 27
- ext,pointTilePlan-method (ext), 13
- ext,spatialTilePlan-method (ext), 13
- ext<- (ext), 13
- ext<-,pointTilePlan,ANY-method (ext), 13
- ext<-,spatialTilePlan,ANY-method (ext), 13
- extending_tilework, 14, 35, 38, 41, 45, 56
- freeTilePlan, 15, 15, 53
- freeTilePlan(), 16, 25, 29, 31, 37, 53, 54
- freeTilePlan-class, 15
- future.apply::future_lapply(), 22, 23
- future::nbrOfWorkers(), 43
- future::plan(), 23, 30

- getBoundedData, 17
- getBoundedData(), 14, 18, 20, 40, 43, 46
- getBoundedData, SpatRaster, numeric-method (getBoundedData), 17
- getBoundedData, SpatRaster, SpatExtent-method (getBoundedData), 17
- getBoundedData, SpatVectorProxy, SpatExtent-method (getBoundedData), 17
- getTile, 18
- getTile(), 14, 16–18, 27–29, 39, 40, 42, 43, 46
- getTile, ANY, tileGroup-method (getTile), 18
- getTile, ANY, tileIterator-method (getTile), 18
- getTile, ANY, tilePlan-method (getTile), 18
- getTile, character, tilePlan-method (getTile), 18
- getTile, SpatRaster, pointTilePlan-method (getTile), 18
- getTile, SpatRaster, tileGroup-method (getTile), 18
- getTile, SpatRaster, tilePlan-method (getTile), 18
- getTile, token, tilePlan-method (getTile), 18
- getTileworkJobID (tilework_management), 55
- getTileworkLogDir (tilework_management), 55
- getTileworkParMethod (parallel_params), 22
- intersect, 4, 5, 7, 9, 10, 12–14, 20, 27
- intersect, freeTilePlan, ANY-method (intersect), 20
- intersect, spatialTilePlan, ANY-method (intersect), 20
- intersect, tilePlan, ANY-method (intersect), 20
- iterSplit, 22
- iterSplit(), 43, 48, 49, 51, 53
- iterSplit, tileIterator-method (iterSplit), 22
- length (dim), 9
- length, tileGroup-method (dim), 9
- length, tileIterator-method (dim), 9
- length, tilePlan-method (dim), 9
- length, tileSelection-method (dim), 9
- length<- (dim), 9
- length<-, spatialTilePlan-method (dim), 9
- ncol (dim), 9
- ncol, tilePlan-method (dim), 9
- nrow (dim), 9
- nrow, tilePlan-method (dim), 9
- parallel_params, 22, 39, 42, 46, 55
- parallelized data processing, 14
- pixelTilePlan, 29, 46, 53, 54
- pixelTilePlan (pixelTilePlan-class), 24
- pixelTilePlan(), 53
- pixelTilePlan-class, 24
- plot, 4, 5, 7, 9, 10, 12–14, 21, 26
- plot, freeTilePlan, missing-method (plot), 26
- plot, pixelTilePlan, missing-method (plot), 26
- plot, pointTilePlan, missing-method (plot), 26
- plot, tilePlan, missing-method (plot), 26
- plot, tileSelection, missing-method (plot), 26
- pointTilePlan, 53
- pointTilePlan (pointTilePlan-class), 27
- pointTilePlan(), 53
- pointTilePlan-class, 27
- quadtreePlan, 30
- quadtreePlan(), 15, 16, 25, 29, 37, 53, 54
- redispatch_tileapply, 38, 41, 45
- redispatch_tileapply (redispatch_tileapply, SpatRaster, freeTilePlan-method), 31
- redispatch_tileapply, ANY, ANY-method (redispatch_tileapply, SpatRaster, freeTilePlan-method), 31
- redispatch_tileapply, ANY, tileGroup-method (redispatch_tileapply, SpatRaster, freeTilePlan-method), 31
- redispatch_tileapply, ANY, tileIterator-method (redispatch_tileapply, SpatRaster, freeTilePlan-method), 31
- redispatch_tileapply, character, tileGroup-method (redispatch_tileapply, SpatRaster, freeTilePlan-method), 31

redispatch_tileapply, character, tileIterator-method, [spatialTilePlan\(\)](#), [53](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), class, [36](#)
[31](#)
 redispatch_tileapply, character, tilePlan-method, terra::Arith-methods, [3](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), [terra::free\(\)](#), [17](#)
[31](#) [terra::window\(\)](#), [17](#)
 redispatch_tileapply, character, tileSelection-method, tilePlanning, [14](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), [tilePlanning\(\)](#), [14](#)
[31](#) [tileApply](#), [40](#), [43](#), [46](#)
 redispatch_tileapply, SpatRaster, freeTilePlan-method, [tileApply\(\)](#), [14](#), [18](#), [20](#), [22](#), [23](#), [40](#), [43](#), [46](#)
[31](#) [tileApply](#), token, missing, tileGroup-method
 redispatch_tileapply, SpatRaster, pixelTilePlan-method (tileApply-group), [38](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), [tileApply](#), token, missing, tileIterator-method
[31](#) (tileApply-iterator), [41](#)
 redispatch_tileapply, SpatRaster, pointTilePlan-method, [tileApply](#), token, missing, tilePlan-method
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), (tileApply-plan), [44](#)
[31](#) [tileApply](#), token, missing, tileSelection-method
 redispatch_tileapply, SpatRaster, spatialTilePlan-method (tileApply-plan), [44](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), [tileApply](#), token, token, tileGroup-method
[31](#) (tileApply-group), [38](#)
 redispatch_tileapply, SpatRaster, tileGroup-method, [tileApply](#), token, token, tileIterator-method
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), (tileApply-iterator), [41](#)
[31](#) [tileApply](#), token, token, tilePlan-method
 redispatch_tileapply, SpatRaster, tileIterator-method (tileApply-plan), [44](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), [tileApply](#), token, token, tileSelection-method
[31](#) (tileApply-plan), [44](#)
 redispatch_tileapply, SpatRaster, tileSelection-method, [tileApply](#)-group, [38](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), [tileApply](#)-iterator, [41](#)
[31](#) [tileApply](#)-plan, [44](#)
 redispatch_tileapply, SpatVector, spatialTilePlan-method, [tileGroup](#), [40](#), [47](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), [tileGroup\(\)](#), [22](#), [40](#), [49](#), [51](#), [53](#)
[31](#) [tileGroup](#)-class, [49](#)
 redispatch_tileapply, SpatVector, tileGroup-method, [tileIterator](#), [43](#), [49](#), [51](#), [53](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), [tileIterator\(\)](#), [22](#), [43](#), [48](#), [49](#), [53](#)
[31](#) [tileIterator](#)-class, [52](#)
 redispatch_tileapply, SpatVector, tileIterator-method, [tilePlan](#), [53](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), [tilePlan\(\)](#), [45](#), [46](#), [25](#), [29](#), [31](#), [37](#), [46](#), [54](#)
[31](#) [tilePlan](#)-class, [54](#)
 redispatch_tileapply, SpatVector, tileSelection-method, [tilePlan](#)-class, [54](#)
 (redispatch_tileapply, SpatRaster, freeTilePlan-method), [tilework](#)-management, [23](#), [55](#)
[31](#) token, [32](#)
 token-class, [55](#)
 setTileworkLogDir
 (tilework_management), [55](#)
 setTileworkParMethod (parallel_params),
[22](#)
 spatialTilePlan, [46](#), [53](#), [54](#)
 spatialTilePlan
 (spatialTilePlan-class), [36](#)