Package 'bitfield'

February 16, 2026

Type Package

Title Handle Bitfields to Record Meta Data

Version 1.0.0

Description Record algorithmic and analytic meta data along a workflow to store that in a bitfield, which can be published alongside any (modelled) data products.

URL https://github.com/bitfloat/bitfield, https://bitfloat.github.io/bitfield/

BugReports https://github.com/bitfloat/bitfield/issues

License GPL (>= 3)

Encoding UTF-8

LazyData true

RoxygenNote 7.3.3

Imports base64enc, checkmate, codetools, dplyr, gh, gitcreds, glue, httr, jsonlite, methods, purrr, rlang, stringr, terra, tibble, tidyr, yaml

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Depends R (>= 4.1.0)

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation no

Author Steffen Ehrmann [aut, cre] (ORCID: <https://orcid.org/0000-0002-2958-0796>)

Maintainer Steffen Ehrmann <steffen.ehrmann@posteo.de>

Repository CRAN

Date/Publication 2026-02-16 19:50:08 UTC

Contents

`bitfield-package` *bitfield: Handle Bitfields to record Meta Data*

Description

The bitfield package provides tools to record analytic and algorithmic meta data or just any ordinary values to store in a bitfield. A bitfield can accompany any (modelled) dataset and can give insight into data quality, provenance, and intermediate values, or can be used to store various output values per observation in a highly compressed form.

Details

The general workflow consists of defining a registry with `bf_registry`, mapping tests to bit-flags with `bf_map`, to encode this with `bf_encode` into an integer value that can be stored and published, or decoded (with `bf_decode`) and re-used in a downstream application. Additional bit-flag protocols can be defined (with `bf_protocol`) and shared as standard with the community via `bf_standards`.

Author(s)

Maintainer, Author: Steffen Ehrmann <steffen.ehrmann@posteo.de>

See Also

- Github project: https://github.com/bitfloat/bitfield
- Report bugs: https://github.com/bitfloat/bitfield/issues

.getDependencies *Identify packages to custom functions*

Description

Identify packages to custom functions

Usage

```
.getDependencies(fun)
```

Arguments

fun	function(...) the custom function in which to identify dependencies.

Value

vector of packages that are required to run the function.

.makeDatacite *Create DataCite-compliant metadata structure*

Description

Create DataCite-compliant metadata structure

Usage

```
.makeDatacite(registry)
```

Arguments

registry	Registry object

Value

List with DataCite-compliant structure

`.makeEncoding` *Determine encoding*

Description

Determine encoding

Usage

```
.makeEncoding(var, type, ...)
```

Arguments

`var`	the variable for which to determine encoding.
`type`	the encoding type for which to determine encoding.
`...`	`list(.)` named list of options to determine encoding, see Details.

Details

Floating-point values are encoded using three fields that map directly to bit sequences. Any numeric value can be written in scientific notation. For example, the decimal 923.52 becomes 9.2352 × 10^2. The same principle applies in binary: the value 101011.101 becomes 1.01011101 × 2^5. This binary scientific notation directly yields the three encoding fields:

- Sign: whether the value is positive or negative (here: positive -> 0)
- Exponent: the power of 2 (here: 5)
- Significand: the fractional part after the leading 1 (here: 01011101)

For background on floating-point representation, see 'Floating Point' by Thomas Finley, or explore encodings interactively at https://float.exposed/.

The allocation of bits across these fields can be adjusted to suit different needs: more exponent bits provide a wider range (smaller minimums and larger maximums), while more significand bits provide finer precision. This package documents bit allocation using the notation [s.e.m], where s = sign bits (0 or 1), e = exponent bits, and m = significand bits.

For non-numeric data (boolean or categorical), the same notation applies with sign and exponent set to 0. A binary flag uses [0.0.1], while a categorical variable with 8 levels requires 3 bits, yielding [0.0.3].

Possible options (. . .) of this function are

- `format`: switch that determines the configuration of the floating point encoding. Possible values are `"half"` [1.5.10], `"bfloat16"` [1.8.7], `"tensor19"` [1.8.10], `"fp24"` [1.7.16], `"pxr24"` [1.8.15], `"single"` [1.8.23] and `"double"` [1.11.52],
- `fields`: list of custom values that control how many bits are allocated to `sign`, `exponent` and `significand` for encoding the numeric values,

- `range`: the ratio between the smallest and largest possible value to be reliably represented (modifies the exponent),
- `decimals`: the number of decimal digits that should be represented reliably (modifies the significand).

In a future version, it should also be possible to modify the bias to focus number coverage to where it's most useful for the data.

Value

list of the encoding values for sign, exponent and significand, and an additional provenance term.

`.toBin` *Make a binary value from an integer*

Description

Make a binary value from an integer

Usage

```
.toBin(x, len = NULL, pad = TRUE)
```

Arguments

`x`	`numeric(.)` numeric vector for which to derive the binary values.
`len`	`integerish(1)` the number of bits used to capture each value. If NULL, computed from the maximum value.
`pad`	`logical(1)` whether to pad the binary values with leading zeros to equal width.

`.toDec` *Make an integer from a binary value*

Description

Make an integer from a binary value

Usage

```
.toDec(x)
```

Arguments

`x`	`character(1)` a binary string (sequence of 0s and 1s) for which to derive the integer.

.updateMD5 *Determine and write MD5 sum*

Description

Determine and write MD5 sum

Usage

```
.updateMD5(x)
```

Arguments

x	registry(1) registry for which to determine the MD5 checksum.

Details

This function follows this algorithm:

- set the current MD5 checksum to NA_character_,
- write the registry into the temporary directory,
- calculate the checksum of this file and finally
- store the checksum in the md5 slot of the registry.

This means that when comparing the MD5 checksum in this slot, one first has to set that value also to NA_character_, otherwise the two values won't coincide.

Value

this function is called for its side-effect of storing the MD5 checksum in the md5 slot of the registry.

.validateProtocol *Validate a bit-flag protocol*

Description

Validate a bit-flag protocol

Usage

```
.validateProtocol(protocol)
```

Arguments

protocol	the protocol to validate

Value

the validated protocol

`.validateToken` *Validate a github token*

Description

This function checks whether the user-provided token is valid for use with this package.

Usage

```
.validateToken(token)
```

Arguments

`token`	`character(1)` github PAT (personal access token).

Value

the validated user token

`bf_analyze` *Analyze encoding options for data*

Description

This function helps you choose appropriate bit allocations for encoding data. It auto-detects the data type and provides relevant analysis:

- Numeric with decimals: trade-offs for floating point encoding, which exponent/significand combinations are adequate for your range and precision requirements.
- Integer: (signed) integer encoding, how many bits are required.
- Factor/character: category/enumeration encoding, which levels are in the data and how many bits are required
- Logical: boolean encoding, do NA values require a second bit.

Usage

```
bf_analyze(
  x,
  range = NULL,
  decimals = NULL,
  min_bits = NULL,
  max_bits = 16L,
  fields = NULL,
  plot = FALSE
)
```

Arguments

x	A numeric, integer, logical, factor, character vector, or single layer SpatRaster to analyze. The type is auto-detected.
range	numeric(2) optional target range c(min, max) to design for (float analysis only). Defaults to the actual data range.
decimals	integer(1) optional decimal places of precision required (float analysis only).
min_bits	integer(1) minimum total bits to display in the Pareto table output. Configurations with fewer bits are hidden. Default is NULL (show all).
max_bits	integer(1) maximum total bits to consider. Defaults to 16.
fields	list optional list specifying which configurations to analyze (float analysis only). See Details.
plot	logical(1) whether to generate a plot (float analysis only). Default is FALSE.

Details

All of this can be applied both to columns in a table or layers in a SpatRaster. Use this before bf_map to understand your encoding options.

Value

An object of class bf_analysis with analysis results.

Float analysis output

For numeric (float) data, the output table shows Pareto-optimal exponent/significand configurations. The columns are:

Exp, Sig, Total Number of exponent bits, significand bits, and their sum. More exponent bits extend the representable range (at the cost of coarser resolution), while more significand bits improve resolution within each exponent band.

Underflow Percentage of data values that fall below the smallest representable positive value. These values are rounded to zero.

Overflow Percentage of data values that exceed the largest representable value. These values are clipped to the maximum.

Changed Percentage of data values that change when encoded and decoded (i.e., that do not survive the round-trip exactly).

Min Res, Max Res Smallest and largest step size between adjacent representable values. In minifloat encoding, resolution varies across the range: small values near zero have fine resolution (small steps), while large values have coarse resolution (large steps). A Max Res of 1.0 means that in the coarsest region, only integer values can be represented – continuous input will be rounded to whole numbers.

RMSE Root mean squared error between original and decoded values, computed over all non-NA data points.

Max Err Largest absolute difference between any original value and its decoded counterpart.

Choosing a configuration

The table only shows Pareto-optimal configurations, i.e., those where no other configuration is strictly better on all quality metrics for the same or fewer total bits. To choose between them:

- Check **Underflow** and **Overflow** first. Non-zero values indicate data loss at the extremes of your range. Adding exponent bits or using the `range` argument to widen the target range can help.
- Compare **RMSE** and **Max Err** to your acceptable precision. If you specified `decimals`, look for configurations where Max Res is at most `10^(-decimals)`.
- If **Max Res** is >= 1, decoded values in the upper range will appear as integers even if the input was continuous. This may or may not be acceptable depending on your application.

Specifying configurations with `fields`

By default, all combinations up to `max_bits` are evaluated and only the Pareto front is shown. Use the `fields` argument to instead compare specific configurations:

- `fields = list(exponent = 4)` shows all significand values paired with 4 exponent bits.
- `fields = list(exponent = c(3, 4), significand = c(5, 4))` compares exp=3/sig=5 and exp=4/sig=4.

Examples

```
# float analysis (numeric with decimals)
bf_analyze(bf_tbl$yield)

# with specific decimal precision requirement
bf_analyze(bf_tbl$yield, decimals = 2)

# design for a larger range than current data
bf_analyze(bf_tbl$yield, range = c(0, 20))
```

```
# with visualization
bf_analyze(bf_tbl$yield, decimals = 2, plot = TRUE)

# compare specific configurations
bf_analyze(bf_tbl$yield, fields = list(exponent = c(2, 3, 4), significand = c(5, 4, 3)))

# show all combinations for a specific exponent
bf_analyze(bf_tbl$yield, fields = list(exponent = 4))

# integer analysis
bf_analyze(as.integer(c(0, 5, 10, 100)))

# category/enum analysis
bf_analyze(bf_tbl$commodity)

# boolean analysis
bf_analyze(c(TRUE, FALSE, TRUE, NA))

# raster with attribute table
library(terra)
r <- rast(nrows = 3, ncols = 3, vals = c(0, 1, 2, 0, 1, 2, 0, 1, 2))
levels(r) <- data.frame(id = 0:2, label = c("low", "medium", "high"))
bf_analyze(r)
```

`bf_decode` *Decode (unpack) a bitfield*

Description

This function takes an integer bitfield and the registry used to build it upstream to decode it into bit representation and thereby unpack the data stored in the bitfield.

Usage

```
bf_decode(x, registry, flags = NULL, envir = NULL, verbose = TRUE)
```

Arguments

`x`	integer table or raster of the bitfield. For registries with a `SpatRaster` template, `x` should be a `SpatRaster`. For registries with a `data.frame` template, `x` should be a `data.frame`.
`registry`	`registry(1)` the registry that should be used to decode the bitfield.
`flags`	`character(.)` the name(s) of flags to extract from this bitfield; leave at `NULL` to extract the full bitfield.

envir — environment(1)
optional environment to store decoded flags as individual objects. If NULL (default), returns results as a list or SpatRaster. Use .GlobalEnv to store flags directly in the workspace.

verbose — logical(1)
whether or not to print the registry legend.

Value

Depending on the registry template type and envir parameter: If envir is NULL, returns a named list with decoded values for table templates, or a multi-layer SpatRaster for raster templates. If envir is specified, stores decoded flags as individual objects in that environment and returns invisible(NULL).

Examples

```
# build registry
reg <- bf_registry(name = "testBF", description = "test bitfield",
                   template = bf_tbl)
reg <- bf_map(protocol = "na", data = bf_tbl, registry = reg, x = commodity)
reg <- bf_map(protocol = "matches", data = bf_tbl, registry = reg,
              x = commodity, set = c("soybean", "maize"), na.val = FALSE)
reg

# encode the flags into a bitfield
field <- bf_encode(registry = reg)
field

# decode (somewhere downstream) - returns a named list
decoded <- bf_decode(x = field, registry = reg)
decoded$na_commodity
decoded$matches_commodity

# alternatively, store directly in global environment
bf_decode(x = field, registry = reg, envir = .GlobalEnv, verbose = FALSE)
na_commodity
matches_commodity

# with raster data
library(terra)
bf_rst <- rast(nrows = 3, ncols = 3, vals = bf_tbl$commodity, names = "commodity")
bf_rst$yield <- rast(nrows = 3, ncols = 3, vals = bf_tbl$yield)

reg <- bf_registry(name = "testBF", description = "raster bitfield",
                   template = bf_rst)
reg <- bf_map(protocol = "na", data = bf_rst, registry = reg, x = commodity)
field <- bf_encode(registry = reg)

# decode back to multi-layer raster
decoded <- bf_decode(x = field, registry = reg, verbose = FALSE)
decoded  # SpatRaster with one layer per flag
```

bf_encode *Encode bit flags into a bitfield*

Description

This function picks up the flags mentioned in a registry and encodes them as integer values.

Usage

```
bf_encode(registry)
```

Arguments

registry	registry(1) the registry that should be encoded into a bitfield.

Value

Depending on the registry template type: a `data.frame` with integer columns (one per 32-bit chunk) if template is a table, or a `SpatRaster` with integer layers if template is a raster.

Examples

```
reg <- bf_registry(name = "testBF", description = "test bitfield",
                   template = bf_tbl)
reg <- bf_map(protocol = "na", data = bf_tbl, registry = reg, x = y)

field <- bf_encode(registry = reg)

# with raster data
library(terra)
bf_rst <- rast(nrows = 3, ncols = 3, vals = bf_tbl$commodity, names = "commodity")
bf_rst$yield <- rast(nrows = 3, ncols = 3, vals = bf_tbl$yield)

reg <- bf_registry(name = "testBF", description = "raster bitfield",
                   template = bf_rst)
reg <- bf_map(protocol = "na", data = bf_rst, registry = reg, x = commodity)

field <- bf_encode(registry = reg)  # returns a SpatRaster
```

bf_export *Export bitfield registries*

Description

Export bitfield registries in DataCite-compliant formats for archiving, sharing, and integration with metadata repositories.

Usage

```
bf_export(registry, format, file = NULL)
```

Arguments

registry	registry(1) Registry object to export.
format	character(1) Export format. One of "json", "xml", "yaml", or "rds".
file	character(1) Optional file path to write the output. If NULL, returns the formatted data.

Value

Exported data as character string for formatted outputs, or the registry object for "rds" format. If `file` is specified, returns invisibly and writes to file.

Examples

```
## Not run:
# Create registry with metadata
auth <- person("Jane", "Smith", email = "jane@example.com",
               comment = c(ORCID = "0000-0000-0000-0000"))
reg <- bf_registry(name = "analysis",
                   description = "Data quality assessment",
                   template = bf_tbl,
                   author = auth)

# Export to different formats
bf_export(registry = reg, format = "json", file = "metadata.json")
bf_export(registry = reg, format = "xml", file = "metadata.xml")
yaml_output <- bf_export(registry = reg, format = "yaml")

## End(Not run)
```

bf_flag *Build a flag*

Description

Convert a flag specification into actual flag values

Usage

```
bf_flag(registry, flag = NULL)
```

Arguments

registry	registry(1) an already defined bitfield registry.
flag	character(1) name of the flag to build.

Details

This function extracts the flag specification, including its test to call it on the data from which the flag shall be created.

Value

vector of the flag values.

Examples

```
reg <- bf_registry(name = "testBF", description = "test bitfield",
                   template = bf_tbl)
reg <- bf_map(protocol = "na", data = bf_tbl, registry = reg,
              x = year)
str(reg@flags)

bf_flag(registry = reg, flag = "na_year")
```

bf_map *Map variables to a bitflag*

Description

This function maps values from a dataset to bit flags that can be encoded into a bitfield.

Usage

```
bf_map(protocol, data, registry, ..., name = NULL, na.val = NULL)
```

Arguments

protocol	character(1) the protocol based on which the flag should be determined, see Details.
data	the object to build bit flags for.
registry	registry(1) an already defined bitfield registry.
...	the protocol-specific arguments for building a bit flag, see Details.
name	character(1) optional flag-name.
na.val	value, of the same encoding type as the flag, that needs to be given, if the test for this flag results in NAs.

Details

protocol can either be the name of an internal item (see bf_pcl), a newly built local protocol (bf_protocol) or one that has been imported from the bitfield community standards repo on github (bf_standards). Any protocol has specific arguments, typically at least the name of the column containing the values to test (x). To make this function as general as possible, all of these arguments are specified via the ... argument of bf_map. Internal protocols are:

- na (x): test whether a variable contains NA-values (*boolean*).
- nan (x): test whether a variable contains NaN-values (*boolean*).
- inf (x): test whether a variable contains Inf-values (*boolean*).
- identical (x, y): element-wise test whether values are identical across two variables (*boolean*).
- range (x, min, max): test whether the values are within a given range (*boolean*).
- matches (x, set): test whether the values match a given set (*boolean*).
- grepl (x, pattern): test whether the values match a given pattern (*boolean*).
- category (x): test whether the values are part of a set of given categories. (*enumeration*).
- case (...): test whether values are part of given cases (*enumeration*).
- nChar (x): count the number of characters of the values (*unsigned integer*).
- nInt (x): count the number of integer digits of the values (*unsigned integer*).
- nDec (x): count the decimal digits of the variable values (*unsigned integer*).
- integer (x, ...): encode values as integer bit-sequence. Accepts raw integer data directly, or numeric data with auto-scaling when range, fields, or decimals are provided. With range = c(min, max) and fields = list(significand = n), values are linearly mapped from [min, max] to [0, 2^n - 1] during encoding and back during decoding. The scaling parameters are stored in provenance for transparent round-trips (*signed integer*).
- numeric (x, ...): encode the numeric value as floating-point bit-sequence (see .makeEncoding for details on the ... argument) (*floating-point*).

Value

an (updated) object of class 'registry' with the additional flag defined here.

Notes

Console output from R classes (such as tibble) often rounds or truncates decimal places, even for ordinary numeric vectors. Internally, R stores numeric values as double-precision floating-point numbers (64 bits, with 52 bits for the significand), providing approximately 16 significant decimal digits ($log10(2^52) = 15.65$). If a bit flag appears inconsistent with the displayed values, verify the full precision using `sprintf("%.16f", values)`. Using more than 16 digits will show additional figures, but these are artifacts of binary-to-decimal conversion and carry no meaningful information.

Examples

```
# first, set up the registry
reg <- bf_registry(name = "testBF", description = "test bitfield",
                   template = bf_tbl)

# then, put the test for NA values together
reg <- bf_map(protocol = "na", data = bf_tbl, registry = reg,
              x = year)

# all the other protocols...
# boolean encoding
reg <- bf_map(protocol = "nan", data = bf_tbl, registry = reg,
              x = y)
reg <- bf_map(protocol = "inf", data = bf_tbl, registry = reg,
              x = y)
reg <- bf_map(protocol = "identical", data = bf_tbl, registry = reg,
              x = x, y = y, na.val = FALSE)
reg <- bf_map(protocol = "range", data = bf_tbl, registry = reg,
              x = yield, min = 10.4, max = 11)
reg <- bf_map(protocol = "matches", data = bf_tbl, registry = reg,
              x = commodity, set = c("soybean", "honey"), na.val = FALSE)
reg <- bf_map(protocol = "grepl", data = bf_tbl, registry = reg,
              x = year, pattern = ".*r", na.val = FALSE)

# enumeration encoding
reg <- bf_map(protocol = "category", data = bf_tbl, registry = reg,
              x = commodity, na.val = 0)
reg <- bf_map(protocol = "case", data = bf_tbl, registry = reg, na.val = 4,
              yield >= 11, yield < 11 & yield > 9, yield < 9 & commodity == "maize")

# integer encoding
reg <- bf_map(protocol = "nChar", data = bf_tbl, registry = reg,
              x = commodity, na.val = 0)
reg <- bf_map(protocol = "nInt", data = bf_tbl, registry = reg,
              x = yield)
reg <- bf_map(protocol = "nDec", data = bf_tbl, registry = reg,
              x = yield)
reg <- bf_map(protocol = "integer", data = bf_tbl, registry = reg,
              x = as.integer(year), na.val = 0L)

# integer encoding with auto-scaling (numeric data mapped to integer range)
dat <- data.frame(density = c(0.5, 1.2, 2.8, 0.0, 3.1))
reg2 <- bf_registry(name = "scaledBF", description = "auto-scaled",
```

```
                    template = dat)
reg2 <- bf_map(protocol = "integer", data = dat, registry = reg2,
               x = density, range = c(0, 3.1),
               fields = list(significand = 5), na.val = 0L)

# floating-point encoding
reg <- bf_map(protocol = "numeric", data = bf_tbl, registry = reg,
              x = yield, decimals = 2)

# finally, take a look at the registry
reg

# alternatively, a raster
library(terra)
bf_rst <- rast(nrows = 3, ncols = 3, vals = bf_tbl$commodity, names = "commodity")
bf_rst$yield <- rast(nrows = 3, ncols = 3, vals = bf_tbl$yield)

reg <- bf_registry(name = "testBF", description = "raster bitfield",
                   template = bf_rst)

reg <- bf_map(protocol = "na", data = bf_rst, registry = reg,
              x = commodity)

reg <- bf_map(protocol = "range", data = bf_rst, registry = reg,
              x = yield, min = 5, max = 11)

reg <- bf_map(protocol = "category", data = bf_rst, registry = reg,
              x = commodity, na.val = 0)
reg
```

`bf_pcl` *Internal bit-flag protocols*

Description

Internal bit-flag protocols

Usage

```
bf_pcl
```

Format

a list containing bit-flag protocols for the internal tests. Each protocol is a list itself with the fields `"name"`, `"version"`, `"extends"`, `"extends_note"`, `"description"`, `"encoding_type"`, `"bits"`, `"requires"`, `"test"`, `"data"` and `"reference"`. For information on how they were set up and how you can set up additional protocols, go to `bf_protocol`.

bf_protocol *Define a new bit-flag protocol*

Description

Define a new bit-flag protocol

Usage

```
bf_protocol(
  name,
  description,
  test,
  example,
  type,
  bits = NULL,
  version = NULL,
  extends = NULL,
  note = NULL,
  author = NULL
)
```

Arguments

name	character(1) simple name of this protocol.
description	character(1) formalised description of the operation in this protocol. It will be parsed with glue and used in the bitfield legend, so can include the test arguments as enbraced expressions.
test	function(...) the function used to compute the bit flag (expressed as character string).
example	list(.) named list that contains all arguments in test as name with values of the correct type.
type	character(1) the encoding type according to which the bit flag is determined. Possible values are bool (for binary flags), enum (for cases), int (for integers) and num (for floating-point encoding).
bits	integer(1) in case the flag requires more bits than the data in example indicate, provide this here.
version	character(1) the version of this protocol according to the *semantic versioning specification*, i.e., of the form X.Y.Z, where X is a major version, Y is a minor version and Z

is a bugfix. For additional details on when to increase which number, study this website.

extends — character(1)
optional protocol name and version that is extended by this protocol.

note — character(1)
note on what the extension adds/modifies.

author — person(.)
to attach a reference to this protocol, please provide here the relevant information about the author(s). If this is not provided, the author "unknown" will be used.

Value

list containing bit-flag protocol

Examples

```
newFlag <- bf_protocol(name = "na",
                       description = "{x} contains NA-values{result}.",
                       test = "function(x) is.na(x = x)",
                       example = list(x = bf_tbl$commodity),
                       type = "bool")
```

---

bf_registry *Initiate a new registry*

Description

Initiate a new registry

Usage

```
bf_registry(
  name,
  description,
  template,
  author = NULL,
  project = NULL,
  license = "MIT"
)
```

Arguments

name — character(1)
the name of the bitfield.

description — character(1)
the description of the bitfield.

template	the data object that serves as a template for the bitfield structure. Can be a `data.frame` (or `tibble`) or a `SpatRaster`. The template determines the output format of `bf_encode` and `bf_decode`.
author	`person(.)` the author(s) involved in the creation of this registry.
project	`list(1)` object created with the function `project` that documents the project metadata.
license	`character(1)` license or rights statement.

Value

an empty registry that captures some metadata of the bitfield, but doesn't contain any flags yet.

Examples

```
auth <- person(given = "Jane", family = "Smith",
               email = "jane@example.com", role = c("cre", "aut"))

proj <- project(title = "example project",
                people = c(person("Jane", "Smith", email = "jane@example.com",
                                  role = "aut"),
                           person("Robert", "Jones", role = c("aut", "cre"))),
                publisher = "example publisher",
                type = "Dataset",
                identifier = "10.5281/zenodo.1234567",
                description = "A comprehensive explanation",
                subject = c("keyword", "subject"),
                license = "CC-BY-4.0")

# with a data.frame template
reg <- bf_registry(name = "currentWorkflow",
                   description = "the registry to my modelling pipeline",
                   template = bf_tbl,
                   author = auth,
                   project = proj)

# with a raster template
library(terra)
bf_rst <- rast(nrows = 3, ncols = 3, vals = 1:9)
reg <- bf_registry(name = "rasterWorkflow",
                   description = "raster-based bitfield",
                   template = bf_rst)
```

`bf_standards` *Handle community standard protocols*

Description

This function allows the user to list, pull or push bit-flag protocols to the bitfloat/standards repository on github

Usage

```
bf_standards(
  protocol = NULL,
  remote = NULL,
  action = "list",
  version = "latest",
  change = NULL,
  token = NULL
)
```

Arguments

protocol	character(1) name of the bit-flag protocol to handle. This is either used to filter the list retrieved from remote, the name of the protocol to pull from github, or the name of the new protocol that should be pushed to github.
remote	character(1) the path in the repo, where the protocol is stored or shall be stored. For instance, to store a protocol in https://github.com/bitfloat/standards/distributions/type/distType.y this should be "distributions/type".
action	character(1) whether to push or pull a protocol, or list the remote contents.
version	character(1) version tag for the protocol, must have a semantic versioning pattern, i.e., MAJOR.MINOR.PATCH.
change	character(1) in case you try to push an updated version of a protocol, you must provide a brief description of what has changed from the current version to this version.
token	character(1) your github personal access token (PAT).

Details

Create a Personal Access Token in your github developer settings (or by running usethis::create_github_token()) and store it with gitcreds::gitcreds_set(). The token must have the scope 'repo' so you can authenticate yourself to pull or push community standards, and will only be accessible to your personal R session.

Value

description

Examples

```
## Not run:
# list all currently available standards
bf_standards()

## End(Not run)
```

`bf_tbl` *Example table*

Description

A 9×5 tibble with a range of example data to showcase functionality of this package.

Usage

```
bf_tbl
```

Format

object of class `tibble` has two columns that indicate coordinates, one column that indicates a crop that is grown there, one column that indicates the yield of that crop there and one column that indicates the year of harvest. All columns contain some sort of deviation that may occur in data.

`print.bf_analysis` *Print method for bf_analysis*

Description

Print method for bf_analysis

Usage

```
## S3 method for class 'bf_analysis'
print(x, min_bits = NULL, ...)
```

Arguments

`x`	bf_analysis object
`min_bits`	minimum total bits to display (overrides value from bf_analyze if provided)
`...`	additional arguments (ignored)

`project` *Create a Project Metadata Object*

Description

Create a Project Metadata Object

Usage

```
project(
  title,
  year = format(Sys.Date(), "%Y"),
  language = "en",
  type,
  author = NULL,
  publisher = NULL,
  identifier = NULL,
  description = NULL,
  subject = NULL,
  contributor = NULL,
  license = NULL,
  funding = NULL,
  version = NULL,
  ...
)
```

Arguments

`title`	`character(1)` project title.
`year`	`character(1)` publication year, defaults to current year.
`language`	`character(1)` primary language, defaults to "en".
`type`	`character(1)` resource type, one of "Dataset", "Software", "Image", "Model", "Text", "Collection", "Other".
`author`	`person(.)` person or organization objects created with `person()`.
`publisher`	`character(1)` name of the publishing entity.
`identifier`	`character(1)` project identifier (e.g., DOI).
`description`	`character(1)` abstract or description.

subject	character(.) keywords or classification codes.
contributor	person(.) additional contributors as person objects.
license	character(1) license or rights statement.
funding	character(.) funding information.
version	character(1) version of the resource.
...	additional metadata elements as name-value pairs.

Value

An object of class "project" with standardized metadata fields.

Examples

```
myProj <- project(title = "example project",
                  author = c(person("Jane", "Smith", email = "jane@example.com",
                                    role = "aut",
                                    comment = c(ORCID = "0000-0001-2345-6789",
                                                affiliation = "University of Example",
                                                ROR = "https://ror.org/05gq02987")),
                             person("Robert", "Jones", role = c("aut", "cre"))),
                  publisher = "example consortium",
                  type = "Dataset",
                  identifier = "10.5281/zenodo.1234567",
                  description = "A comprehensive explanation",
                  subject = c("keyword", "subject"),
                  license = "CC-BY-4.0")
```

---

registry-class *Bit registry class (S4) and methods*

Description

A `registry` stores metadata and flag configuration of a bitfield.

Slots

`name` character(1)
: short name of the bitfield.

`version` character(1)
: automatically created version tag of the bitfield. This consists of the package version, the version of R and the date of creation of the bitfield.

`md5` `character(1)`
: the MD5 checksum of the bitfield as determined with md5sum.

`description` `character(1)`
: longer description of the bitfield.

`template` `list(.)`
: structural metadata for encoding/decoding, including: `type` ("data.frame" or "SpatRaster"), `width` (total bits), `length` (number of observations/cells), and for rasters: `nrows`, `ncols`, `extent`, `crs`.

`flags` `list(.)`
: list of flags in the registry.

`show,registry-method` *Print registry in the console*

Description

Print registry in the console

Usage

```
## S4 method for signature 'registry'
show(object)
```

Arguments

`object`	`registry(1)` object to show.

Details

This method produces an overview of the registry by printing a header with information about the setup of the bitfield and a table with one line for each flag in the bitfield. The table shows the start position of each flag, the encoding type (see `.makeEncoding`), the bitfield operator type and the columns that are tested by the flag.

Index