Package 'bitfield'

July 22, 2025

Type Package

Title Handle Bitfields to Record Meta Data

Version 0.6.1

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 7.3.2

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

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Depends R (>= 4.1.0)

Config/testthat/edition 3

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 2025-05-01 10:30:08 UTC

Contents

bitfield-package . . . 2
.getDependencies . . . 3
.makeEncoding . . . 3
.rast . . . 4
.toBin . . . 5
.toDec . . . 6
.updateMD5 . . . 6
.validateProtocol . . . 7
.validateToken . . . 7
bf_decode . . . 8
bf_encode . . . 9
bf_map . . . 9
bf_pcl . . . 12
bf_protocol . . . 12
bf_registry . . . 14
bf_standards . . . 14
bf_tbl . . . 15
registry-class . . . 16
show,registry-method . . . 17

Index **18**

`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.

`.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 with three fields that can be readily stored as bit sequence. Any numeric value can be represented in scientific notation, for example, the decimal 923.52 can be represented as 9.2352 * 10^2. These decimal values can be transformed to binary values, which can then likewise be represented in scientific notation. Here, the 10 is replaced by a 2 (because we go from decimal to binary), for example the binary value 101011.101 can be represented as 1.01011101 * 2^5. This scientific notation can now be broken down into the three previously mentioned fields, one for the sign (positive or negative), one for the exponent and one for the remaining part, the mantissa (or significand). For background information on how these fields are processed, study for instance 'Floating Point' by Thomas Finley and check out https://float.exposed/ to play around with floating point encoding. Depending on the encoding needs, these three values can be adapted, for example increase the exponent to provide a wider range (i.e., smaller small and larger large values) or increase the mantissa to provide more precision (i.e., more decimal digits). In the scope of this package, these three values are documented with a tag of the form [x.y.z], with x = number of sign bits (either 0 or 1), y = number of exponent bits, and z number of mantissa bits.

When handling values that are not numeric, this package makes use of the same system, only that sign and exponent are set to 0, while the mantissa bits are set to either 1 (for binary responses [0.0.1]), or to whatever number of cases are required (i.e., for 8 cases with 3 required bits, resulting in the tag [0.0.3]).

Possible options (. . .) of this function are

- `precision`: 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 `mantissa` 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 mantissa).

Value

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

`.rast` *Extract values and metadata from terra::SpatRaster*

Description

Extract values and metadata from terra::SpatRaster

Usage

```
.rast(x)
```

Arguments

x	SpatRaster(1) the SpatRaster object.

Details

This function simply extracts the values from x and appends the raster metadata as attributes.

Value

the function that extracts values and metadata.

.toBin *Make a binary value from an integer*

Description

Make a binary value from an integer

Usage

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

Arguments

x	numeric(1) the numeric value for which to derive the binary value.
len	integerish(1) the number of bits used to capture the value.
pad	logical(1) whether to pad the binary value with 0 values.
dec	logical(1) whether to transform the decimal part to bits, or the integer part.

.toDec *Make an integer from a binary value*

Description

Make an integer from a binary value

Usage

```
.toDec(x)
```

Arguments

x	character(1) the binary value (character 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_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, sep = NULL, verbose = TRUE)
```

Arguments

x	integerish(1) table of the integer representation of the bitfield.
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.
sep	character(1) a symbol with which, if given, the distinct fields shall be separated.
verbose	logical(1) whether or not to return the registry legend.

Value

data.frame with the binary values of flags in the registry in columns.

Examples

```
# build registry
reg <- bf_map(protocol = "na", data = bf_tbl, x = commodity)
reg <- bf_map(protocol = "matches", data = bf_tbl, x = commodity, set = c("soybean", "maize"),
              registry = reg)
reg

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

# decode (somewhere downstream)
flags <- bf_decode(x = field, registry = reg, sep = "-")
flags

# more reader friendly
cbind(bf_tbl, bf_decode(x = field, registry = reg, verbose = FALSE))
```

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

data.frame of the same length as the input data. Depending on type and amount of bit flags, this can a table with any number of columns, each of which encodes a sequence of 32 bits into an integer.

Examples

```
reg <- bf_map(protocol = "na", data = bf_tbl, x = y)

field <- bf_encode(registry = reg)
```

bf_map *Build a bit flag*

Description

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

Usage

```
bf_map(
  protocol,
  data,
  ...,
  name = NULL,
  pos = NULL,
  na.val = NULL,
  description = NULL,
  registry = 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.
...	the protocol-specific arguments for building a bit flag, see Details.
name	character(1) optional flag-name.
pos	integerish(.) the position(s) in the bitfield that should be set.
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.
description	character(.) optional description that should be used instead of the default protocol-specific description. This description is used in the registry legend, so it should have as many entries as there will be flags (two for a binary flag, as many as there are cases for a enumeration flag and one for integer or numeric flags).
registry	registry(1) a bitfield registry that has been defined with bf_registry; if it's undefined, an empty registry will be defined on-the-fly.

Details

protocol can either be the name of an internal item bf_pcl, a newly built local protocol or one that has been imported from the bitfield community standards repo on github. Any protocol has specific arguments, typically at least the name of the column containing the variable values (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*).
- 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 the integer values as bit-sequence (*signed integer*).
- numeric (x, ...): encode the numeric value as floating-point bit-sequence (with an adapted precision) (*floating-point*).

Value

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

Notes

The console output of various classes (such as tibble) shows decimals that are not present or rounds decimals that are present, even for ordinary numeric vectors. R stores numeric values internally as double-precision floating-point values (with 64 bits, where 52 bits encode the mantissa), which corresponds to a decimal precision of ~16 digits (`log10(2^52)`). Hence, if a bit flag doesn't seem to coincide with the values you see in the console, double check the values with `sprintf("%16f", values)`. If you use a larger decimal precision, you'll see more digits, but those are not meaningful, as they result merely from the binary-to-decimal conversion (check out `.makeEncoding` for additional information.

When testing for cases, they are evaluate in the order they have been defined in. If an observation is part of two cases, it will thus have the value of the last case it matches. The encoding type of cases is given as *enumeration*, which means that the values can be either `integer` or `factor`. Both are handled as if they were integers internally, so even though an enumeration data type could in principle also be a `character`, this is possible within the scope of this package. Bitflag protocols that extend the *case* protocol must thus always result in integer values.

Examples

```
opr <- "identical"

# identify which arguments need to be given to call a test ...
formalArgs(bf_pcl[[opr]]$test)

# put the test together
bf_map(protocol = opr, data = bf_tbl, x = x, y = y, na.val = FALSE)

# some other examples of ...
# boolean encoding
bf_map(protocol = "matches", data = bf_tbl, x = commodity, set = c("soybean", "honey"))
bf_map(protocol = "range", data = bf_tbl, x = yield, min = 10.4, max = 11)

# enumeration encoding
bf_map(protocol = "case", data = bf_tbl,
        yield >= 11, yield < 11 & yield > 9, yield < 9 & commodity == "maize")

# integer encoding
bf_map(protocol = "integer", data = bf_tbl, x = as.integer(year), na.val = 0L)

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

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 build the bit flag.
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 = NULL, description = NULL)
```

Arguments

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

Value

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

Examples

```
reg <- bf_registry(name = "currentWorkflow",
                   description = "this is to document my current workflow so
                                  that I can share it with my colleagues
                                  alongside a publication.")
```

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 10×5 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.

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

Description

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

Slots

`width` `integerish(1)`
: how many bits is the bitfield wide.

`length` `integerish(1)`
: how many observations are encoded in the bitfield.

`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.

`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

∗ **datasets**
    bf_pcl, 12
    bf_tbl, 15
.getDependencies, 3
.makeEncoding, 3, *11*, *17*
.rast, 4
.toBin, 5
.toDec, 6
.updateMD5, 6
.validateProtocol, 7
.validateToken, 7

bf_decode, *2*, 8
bf_encode, *2*, 9
bf_map, *2*, 9
bf_pcl, *10*, 12
bf_protocol, *2*, *12*, 12
bf_registry, *2*, *10*, 14
bf_standards, *2*, 14
bf_tbl, 15
bitfield *(*bitfield-package*)*, 2
bitfield-package, 2

character(.), *8*, *10*
character(1), *6–8*, *10*, *13–16*

function(...), *3*, *13*

glue, *13*

integer(1), *13*
integerish(.), *10*
integerish(1), *5*, *8*, *16*

list(.), *3*, *13*, *16*
logical(1), *5*, *8*

md5sum, *16*

numeric(1), *5*

person(.), *13*

registry *(*registry-class*)*, 16
registry(1), *6*, *8–10*, *17*
registry-class, 16

show,registry-method, 17
SpatRaster(1), *5*