Package: ids (via r-universe)

September 13, 2024

Title Generate Random Identifiers

Version 1.2.2

Description Generate random or human readable and pronounceable identifiers.

License MIT + file LICENSE

URL https://github.com/reside-ic/ids, https://reside-ic.github.io/ids/

BugReports https://github.com/reside-ic/ids/issues

Suggests knitr, mockery, openssl (>= 0.9.6), rcorpora, rmarkdown, testthat (>= 3.0.0)

RoxygenNote 7.1.2

Roxygen list(markdown = TRUE)

VignetteBuilder knitr

Encoding UTF-8

Language en-GB

Config/testthat/edition 3

Repository https://reside-ic.r-universe.dev

RemoteUrl https://github.com/reside-ic/ids

RemoteRef HEAD

RemoteSha 07e73255a5cfeb3aea06a0bf387b546e725570af

Contents

`adjective_animal` *Ids based on a number of adjectives and an animal*

Description

Ids based on a number of adjectives and an animal

Usage

```
adjective_animal(
  n = 1,
  n_adjectives = 1,
  style = "snake",
  max_len = Inf,
  alliterate = FALSE,
  global = TRUE,
  use_openssl = NULL
)
```

Arguments

`n`	number of ids to return. If `NULL`, it instead returns the generating function
`n_adjectives`	Number of adjectives to prefix the animal with
`style`	Style to join words with. Can be one of "Pascal", "camel", "snake", "kebab", "dot", "title", "sentence", "lower", "upper", "constant" or "spongemock"
`max_len`	The maximum length of a word part to include (this may be useful because some of the names are rather long. This stops you generating a `hexakosioihexekontahexaphobic_queenalex`. A vector of length 2 can be passed in here in which case the first element will apply to the adjectives (all of them) and the second element will apply to the animals.
`alliterate`	Produce "alliterative" adjective animals (e.g., `hessian_hamster`). Note that this cannot provide an equal probability of any particular combination because it forces a weighted sampling. Adjectives may also be repeated if `n_adjectives` is more than 1.
`global`	Use global random number generator that responds to `set.seed` (see random_id for details, but note that the default here is different).
`use_openssl`	Use openssl for random number generation when using a non-global generator (see random_id for details)

Details

The list of adjectives and animals comes from https://github.com/a-type/adjective-adjective-animal, and in turn from <gfycat.com>

Author(s)

Rich FitzJohn

Examples

```
# Generate a random identifier:
ids::adjective_animal()

# Generate a bunch all at once:
ids::adjective_animal(5)

# Control the style of punctuation with the style argument:
ids::adjective_animal(style = "lower")
ids::adjective_animal(style = "CONSTANT")
ids::adjective_animal(style = "camel")
ids::adjective_animal(style = "kebab")
ids::adjective_animal(style = "spongemock")

# Control the number of adjectives used
ids::adjective_animal(n_adjectives = 3)

# This can get out of hand quickly though:
ids::adjective_animal(n_adjectives = 7)

# Limit the length of adjectives and animals used:
ids::adjective_animal(10, max_len = 6)

# The lengths can be controlled for adjectives and animals
# separately, with Inf meaning no limit:
ids::adjective_animal(10, max_len = c(6, Inf), n_adjectives = 2)

# Pass n = NULL to bind arguments to a function
id <- ids::adjective_animal(NULL, n_adjectives = 2,
                            style = "dot", max_len = 6)
id()
id(10)

# Alliterated adjective animals always aid added awesomeness
ids::adjective_animal(10, n_adjectives = 3, alliterate = TRUE)
```

`ids` *Generic id generating function*

Description

Generic id generating function

Usage

```
ids(
  n,
  ...,
  vals = list(...),
  style = "snake",
  global = TRUE,
  use_openssl = FALSE
)
```

Arguments

n	number of ids to return. If NULL, it instead returns the generating function
...	A number of character vectors
vals	A list of character vectors, *instead* of ...
style	Style to join words with. Can be one of "Pascal", "camel", "snake", "kebab", "dot", "title", "sentence", "lower", "upper", "constant" or "spongemock"
global	Use global random number generator that responds to set.seed (see random_id for details, but note that the default here is different).
use_openssl	Use openssl for random number generation when using a non-global generator (see random_id for details)

Value

Either a character vector of length n, or a function of one argument if n is NULL

Author(s)

Rich FitzJohn

Examples

```
# For an example, please see the vignette
```

---

int_to_proquint *Convert to and from proquints*

Description

Convert to and from proquints.

Usage

```
int_to_proquint(x, use_cache = TRUE)

proquint_to_int(p, as = "numeric", use_cache = TRUE)

proquint_word_to_int(w, use_cache = TRUE, validate = TRUE)

int_to_proquint_word(i, use_cache = TRUE, validate = TRUE)
```

Arguments

x	An integer (or integer-like) value to convert to a proquint
use_cache	Because there are relatively few combinations per word, and because constructing short strings is relatively expensive in R, it may be useful to cache all 65536 possible words. If TRUE then the first time that this function is used all words will be cached and the results used - the first time may take up to ~1/4 of a second and subsequent calls will be much faster. The identifiers selected will not change with this option (i.e., given a particular random seed, changing this option will not affect the identifiers randomly selected).
p	A character vector representing a proquint
as	The target data type for conversion from proquint. The options are integer, numeric and bignum. The first two will overflow given sufficiently large input - this will throw an error (overflow is at .Machine$integer.max and 2 / .Machine$double.eps - 1 for integer and numeric respectively). For bignum this will return a *list* of bignum elements *even if* p *is of length 1*.
w	A proquint *word* (five letter string)
validate	Validate the range of inputs? Because these functions are used internally, they can skip input validation. You can too if you promise to pass sanitised input in. If out-of-range values are passed in and validation is disabled the behaviour is undefined and subject to change.
i	An integer representing a single proquint word (in the range 0:65535)

Details

These functions try to be type safe and predictable about what they will and will not return.

For proquint_to_int, because numeric overflow is a possibility, it is important to consider whether a proquint can be meaningfully translated into an integer or a numeric and the functions will throw an error rather than failing in a more insidious way (promoting the type or returning NA).

proquint_word_to_int always returns an integer vector of the same length as the input.

Missing values are allowed; a missing integer representation of a proquint will translate as NA_character_ and a missing proquint will translate as NA_integer_ (if as = "integer"), NA_real_, if as = "numeric" or as NULL (if as = "bignum").

Names are always discarded. Future versions may gain an argument named with a default of FALSE, but that setting to TRUE would preserve names. Let me know if this would be useful.

`proquint` *Generate random proquint identifiers*

Description

Generate random "proquint" identifiers. "proquint" stands for PRO-nouncable QUINT-uplets and were described by Daniel Wilkerson in https://arxiv.org/html/0901.4016. Each "word" takes one of `2^16` possibilities. A four word proquint has a keyspace of `10^19` possibilities but takes only 23 characters. Proquint identifiers can be interchanged with integers (though this is totally optional); see proquint_to_int and the other functions documented on that page.

Usage

```
proquint(
  n = 1,
  n_words = 2L,
  use_cache = TRUE,
  global = TRUE,
  use_openssl = NULL
)
```

Arguments

`n`	number of ids to return. If `NULL`, it instead returns the generating function
`n_words`	The number of words for each identifier; each word has `2^16` (65536) possible combinations, a two-word proquint has `2^32` possible combinations and an k-word proquint has `2^(k * 16)` possible combinations.
`use_cache`	Because there are relatively few combinations per word, and because constructing short strings is relatively expensive in R, it may be useful to cache all 65536 possible words. If `TRUE` then the first time that this function is used all words will be cached and the results used - the first time may take up to ~1/4 of a second and subsequent calls will be much faster. The identifiers selected will not change with this option (i.e., given a particular random seed, changing this option will not affect the identifiers randomly selected).
`global`	Use global random number generator that responds to `set.seed` (see random_id for details, but note that the default here is different).
`use_openssl`	Use openssl for random number generation when using a non-global generator (see random_id for details)

Details

In the abstract of their paper, Wilkerson introduces proquints:

"Identifiers (IDs) are pervasive throughout our modern life. We suggest that these IDs would be easier to manage and remember if they were easily readable, spellable, and pronounceable. As a solution to this problem we propose using PRO-nouncable QUINT-uplets of alternating unambiguous consonants and vowels: proquints."

Examples

```
# A single, two word, proquint
ids::proquint()

# Longer identifier:
ids::proquint(n_words = 5)

# More identifiers
ids::proquint(10)
```

`random_id` *Random hexadecimal identifiers*

Description

Random hexadecimal identifiers. If possible, by default this uses the `openssl` package to produce a random set of bytes, and expresses that as a hex character string, creating cryptographically secure (unpredictable) identifiers. If that is unavailable, fall back on the xoshiro128+ algorithm to produce random numbers that are not cryptographically secure, but which do not affect the global random number stream (see Details). If desired, you can produce "predictable" random identifiers that respect the value of the global random number stream via `set.seed`.

Usage

```
random_id(n = 1, bytes = 16, use_openssl = NULL, global = FALSE)
```

Arguments

`n`	number of ids to return. If `NULL`, it instead returns the generating function
`bytes`	The number of bytes to include for each identifier. The length of the returned identifiers will be twice this long with each pair of characters representing a single byte.
`use_openssl`	Optionally a logical, indicating if we should use the openssl for generating the random identifiers from the non-global source. If not given we prefer to use openssl if it is available but fall back on R (See Details).
`global`	Logical, indicating if random numbers should be global (given R's global random number seed). If `TRUE`, then ids generated will be predictable.

Details

Since ids version 1.2.0, the openssl package is optional, and this affects non-global random number drawing. If you have openssl installed your random numbers will be ~50x faster than the implementation we include here.

If `global = TRUE` we always use a simple sample based algorithm that is driven from the global random number stream. However, when `global = FALSE` the behaviour depends on the value of `use_openssl` and whether that package is installed, either using the openssl generators, using an internal algorithm based on xoshiro128+ or erroring.

- use_openssl = NULL and openssl installed: openssl
- use_openssl = NULL and openssl missing: internal
- use_openssl = TRUE and openssl installed: openssl
- use_openssl = TRUE and openssl missing: error
- use_openssl = FALSE: internal

Author(s)

Rich FitzJohn

Examples

```
# Generate a random id:
ids::random_id()

# Generate 10 of them!
ids::random_id(10)

# Different length ids
random_id(bytes = 8)
# (note that the number of characters is twice the number of bytes)

# The ids are not affected by R's RNG state:
set.seed(1)
(id1 <- ids::random_id())
set.seed(1)
(id2 <- ids::random_id())
# The generated identifiers are different, despite the seed being the same:
id1 == id2

# If you need these identifiers to be reproducible, pass use_openssl = FALSE
set.seed(1)
(id1 <- ids::random_id(use_openssl = FALSE))
set.seed(1)
(id2 <- ids::random_id(use_openssl = FALSE))
# This time they are the same:
id1 == id2

# Pass `n = NULL` to generate a function that binds your arguments:
id8 <- ids::random_id(NULL, bytes = 8)
id8(10)
```

---

sentence *Sentence style identifiers*

Description

Create a sentence style identifier. This uses the approach described by Asana on their blog https://blog.asana.com/2011/09/6-sad-squid-snuggle-softly/. This approach encodes 32 bits of information (so 2^32 ~= 4 billion possibilities) and in theory can be remapped to an integer if you really wanted to.

Usage

```
sentence(
  n = 1,
  style = "snake",
  past = FALSE,
  global = TRUE,
  use_openssl = NULL
)
```

Arguments

n	number of ids to return. If NULL, it instead returns the generating function
style	Style to join words with. Can be one of "Pascal", "camel", "snake", "kebab", "dot", "title", "sentence", "lower", "upper", "constant" or "spongemock"
past	Use the past tense for verbs (e.g., slurped or jogged rather than slurping or jogging)
global	Use global random number generator that responds to set.seed (see random_id for details, but note that the default here is different).
use_openssl	Use openssl for random number generation when using a non-global generator (see random_id for details)

Author(s)

Rich FitzJohn

Examples

```
# Generate an identifier
ids::sentence()

# Generate a bunch
ids::sentence(10)

# As with adjective_animal, use "style" to control punctuation
ids::sentence(style = "Camel")
ids::sentence(style = "dot")
ids::sentence(style = "Title")

# Change the tense of the verb:
set.seed(1)
ids::sentence()
set.seed(1)
```

```
ids::sentence(past = TRUE)

# Pass n = NULL to bind arguments to a function
id <- ids::sentence(NULL, past = TRUE, style = "dot")
id()
id(10)
```

---

uuid *Generate UUIDs*

Description

Generate UUIDs (Universally Unique IDentifiers). In previous versions this was simply a thin wrapper around `uuid::UUIDgenerate`, however this was subject to collisions on windows where relatively small numbers of UUIDs generated at the same time could return values that were identical. We now generate only version 4 UUIDs (i.e., random though with particular bits set).

Usage

```
uuid(n = 1, drop_hyphens = FALSE, use_time = NA)
```

Arguments

`n`	number of ids to return. If `NULL`, it instead returns the generating function
`drop_hyphens`	Drop the hyphens from the UUID?
`use_time`	Unused.

Author(s)

Rich FitzJohn

Examples

```
# Generate one id
ids::uuid()

# Or a bunch
ids::uuid(10)

# More in the style of random_id()
ids::uuid(drop_hyphens = TRUE)
```

Index