Package 'edm1'

October 23, 2024

Title Simplify Complex Data Manipulation

Version 2.0.0.0

Description Provides complex sorting algorythms. Provides date manipulation algorythms. In addition to providing handy functions to discretize variables, an SQL joins alternatives, a set of function to work with geographical coordinates, and other functions to work with text mining.
License GPL (==3)
Encoding UTF-8
Roxygen list(markdown = TRUE)
RoxygenNote 7.3.1
Imports stringr, stringi, dplyr, data.table, openxlsx
Depends stringr, stringi, dplyr, data.table, openxlsx Contents
all_concat all_stat and_bool1 and_bool2 any_join_datf appndr 1 arroundr_mean 1 arroundr_min 1 better_match 1 better_split 1 better_split_any 1 better_sub 1 better_sub_mult 1 better_unique 1 bind_cols 1

2 Contents

oind_rows	18
an_be_num	18
closer_ptrn	19
eloser_ptrn_adv	21
·lusterizer_v	23
colins_datf	25
ol_convertr	26
col_to_row	27
onverter_date	27
onverter_format	28
cost_and_taxes	29
rumulated rows	30
rumulated_rows_na	31
 cutr_v	31
 zut_v	32
lata_gen	33
lata_meshup	35
late_addr	36
late_converter_reverse	37
latf_appendr	38
latf_appendr2	38
_ 11	39
latf_insertr	39 40
11	
latf_row_appendr2	40
lcr_untl	41
lcr_val	42
lelta_normal	42
lelta_unif	43
lepth_pairs_findr	
liff_datf	
lynamic_idx_convertr	45
dm1_random_val	
dm1_random_val_spe	
dm1_rnorm1	
dm1_rnorm2	
dm1_runif_deterministic	51
dm1_unif_time	
dm_arrangr	53
dm_arrangr2	54
dm_group_by1	55
cdm_group_by2	56
dm_pert	57
dm_pivot_longer1	58
edm_pivot_longer2	61
edm pivot series	64
	66
prot_wider2	67
elements_equalifier	69
equalizer_v	70
extract_normal	70
extrt_only_v	76
311r	70

Contents 3

4 Contents

nestr_datf2	. 114
nest_v	. 114
new_ordered	. 115
normal dens	. 116
normal offset prob	. 116
normal_offset_val	. 117
occu	
old_to_new_idx	
or_bool1	
or_bool2	
pairs_findr	
pairs_findr_merger	
pairs_insertr	
pairs_insertr2	
paste_datf	
paste_datf2	
paste_datt2	
-e	
pattern_gettr	
pattern_tuning	
power_to_char	
pre_to_post_idx	
ptrn_switchr	
ptrn_twkr	
read_edm_parser	
rearangr_v	
regex_spe_detect	. 132
regroupr	. 133
rm_na_rows	. 134
rm_rows	. 135
row_to_col	. 136
r_print	. 136
save untl	. 137
see_datf	. 138
see diff	
see_diff_all	
see_diff_detailled	
see_file	
see_idx	
see_inside	
see_in_grep	
see_in_l	
see_mode	
selected_char	
sequence_na_mean1	
sequence_na_mean2	
sequence_na_med1	
sequence_na_med2	
sort_date	
sort_normal_qual	
sort_normal_qual2	
split_by_step	
str_remove_unt1	158

all_concat 5

	sub_mult	159
	successive_diff	160
	sum_group1	160
	sum_group2	162
	swipr	164
	test_order	165
	time_serie_equalizer	165
	to_unique	167
	union_all	168
	union_keep	168
	unique_datf	169
	unique_ltr_from_v	170
	unique_pos	
	unique_total	
	until_stnl	
	val_replacer	
	vector_replacor	172
	vec_in_datf	
	vlookup_datf	
	wider_datf	
	wide_to_narrow_idx	
	write_edm_parser	
	1	
Index		178
2]]	concat all concat	
a_1_	_COITCAC	

Description

Output all the possible concatenations of elements in different vectors, see examples

Usage

```
all\_concat(..., sep = "\_")
```

Arguments

is all the vectors of the elements that will be concatenated, see examplesis the separator to use between concatenated elements

```
print(all_concat(c("France", "Germany"), c("2012", "2013"), c(1:2), sep = "_"))

[1] "France_2012_1" "Germany_2012_1" "France_2013_1" "Germany_2013_1"

[5] "France_2012_2" "Germany_2012_2" "France_2013_2" "Germany_2013_2"
```

6 all_stat

all_stat

Description

Allow to see all the main statistics indicators (mean, median, variance, standard deviation, sum, max, min, quantile) of variables in a dataframe by the modality of a variable in a column of the input datarame. In addition to that, you can get the occurence of other qualitative variables by your chosen qualitative variable, you have just to precise it in the vector "stat_var" where all the statistics indicators are given with "occu-var_you_want/".

Usage

```
all_stat(inpt_v, var_add = c(), stat_var = c(), inpt_datf)
```

Arguments

inpt_v is the modalities of the variables
var_add is the variables you want to get the stats from

stat_var is the stats indicators you want

inpt_datf is the input dataframe

```
datf <- data.frame("mod"=c("first", "seco", "seco", "first", "first", "third", "first"),</pre>
                "var1"=c(11, 22, 21, 22, 22, 11, 9),
               "var2"=c("d", "d", "z", "z", "z", "d",
               "var3"=c(45, 44, 43, 46, 45, 45, 42),
              "var4"=c("A", "A", "A", "A", "B", "C", "C"))
print(all_stat(inpt_v=c("first", "seco"), var_add = c("var1", "var2", "var3", "var4"),
 stat_var=c("sum", "mean", "median", "sd", "occu-var2/", "occu-var4/", "variance",
"quantile-0.75/"),
 inpt_datf=datf))
#
    modal_v var_vector occu sum mean med standard_devaition
                                                                       variance
#1
     first
#2
                              64
                                   16 16.5
                                             6.97614984548545 48.6666666666667
                  var1
#3
                var2-d
                          1
#4
                var2-z
#5
                            178 44.5
                                       45
                                             1.73205080756888
                                                                               3
                  var3
#6
                var4-A
                          2
#7
                var4-B
                          1
#8
                var4-C
                          1
#9
       seco
#10
                              43 21.5 21.5 0.707106781186548
                                                                            0.5
                  var1
#11
                var2-d
                          1
#12
                var2-z
                          1
#13
                              87 43.5 43.5 0.707106781186548
                                                                            0.5
                  var3
#14
                var4-A
                          2
#15
                var4-B
                          0
#16
                var4-C
    quantile-0.75
```

and_bool1 7

#1	
#2	22
#3	
#4	
#5	45.25
#6	
#7	
#8	
#9	
#10	21.75
#11	
#12	
#13	43.75
#14	
#15	
#16	

and_bool1	and_bool1
-----------	-----------

Description

Returns a boolean vector according to 'and' condition on boolean vector in a dataframe, each column represents a boolean vector, see examples

Usage

```
and_bool1(inpt_datf)
```

Examples

```
and_bool2 and_bool2
```

Description

Returns a boolean vector according to 'and' condition on boolean vector in a dataframe, each column represents a boolean vector, see examples, uses a different algorithm than and_bool1

Usage

```
and_bool2(inpt_datf)
```

8 any_join_datf

Examples

any_join_datf

any_join_datf

Description

Allow to perform SQL joints with more features

Usage

```
any_join_datf(
  inpt_datf_l,
  join_type = "inner",
  join_spe = NA,
  id_v = c(),
  excl_col = c(),
  rtn_col = c(),
  d_val = NA
)
```

Arguments

inpt_datf_l	is a list containing all the dataframe
join_type	is the joint type. Defaults to inner but can be changed to a vector containing all the dataframes you want to take their ids to don external joints.
join_spe	can be equal to a vector to do an external joints on all the dataframes. In this case, join_type should not be equal to "inner"
id_v	is a vector containing all the ids name of the dataframes. The ids names can be changed to number of their columns taking in count their position in inpt_datf_l. It means that if my id is in the third column of the second dataframe and the first dataframe have 5 columns, the column number of the ids is $5 + 3 = 8$
excl_col	is a vector containing the column names to exclude, if this vector is filled so "rtn_col" should not be filled. You can also put the column number in the manner indicated for "id_v". Defaults to $c()$
rtn_col	is a vector containing the column names to retain, if this vector is filled so "excl_col" should not be filled. You can also put the column number in the manner indicated for "id_v". Defaults to $c()$
d_val	is the default val when here is no match

any_join_datf 9

```
datf1 <- data.frame("val"=c(1, 1, 2, 4), "ids"=c("e", "a", "z", "a"),</pre>
"last"=c("oui", "oui", "non", "oui"),
"second_ids"=c(13, 11, 12, 8), "third_col"=c(4:1))
datf2 \leftarrow data.frame("val"=c(3, 7, 2, 4, 1, 2), "ids"=c("a", "z", "z", "a", "a", "a"),
"bool"=c(TRUE, FALSE, FALSE, FALSE, TRUE, TRUE),
"second_ids"=c(13, 12, 8, 34, 22, 12))
datf3 <- data.frame("val"=c(1, 9, 2, 4), "ids"=c("a", "a", "z", "a"),</pre>
"last"=c("oui", "oui", "non", "oui"),
"second_ids"=c(13, 11, 12, 8))
print(any_join_datf(inpt_datf_l=list(datf1, datf2, datf3), join_type="inner",
id_v=c("ids", "second_ids"),
              excl_col=c(), rtn_col=c()))
# ids val ids last second_ids val ids bool second_ids val ids last second_ids
#3 z12 2 z non 12 7 z FALSE 12 2 z non 12
print(any_join_datf(inpt_datf_l=list(datf1, datf2, datf3), join_type="inner", id_v=c("ids
excl_col=c(), rtn_col=c()))
# ids val ids last second_ids val ids bool second_ids val ids last second_ids
#2 a 1 a oui 11 3 a TRUE 13 1 a oui
                       12 7 z FALSE
                                            12 2 z non
   Z
          z non
#3
                                                                  12
                       8 4 a FALSE
      4 a oui
   а
                                            34
                                                9 a oui
#4
                                                                 11
print(any_join_datf(inpt_datf_l=list(datf1, datf2, datf3), join_type=c(1), id_v=c("ids"),
              excl_col=c(), rtn_col=c()))
# ids val ids last second_ids val ids bool second_ids val ids last
11 3 a TRUE
#2 a 1 a oui
                                              13 1 a oui
#3 z 2 z non
                       12 7 z FALSE
                                              12 2 z non
                      8 4 a FALSE
#4 a 4 a oui
                                              34 9 a oui
# second_ids
#1 <NA>
#2
        13
#3
         12
#4
         11
print(any_join_datf(inpt_datf_l=list(datf2, datf1, datf3), join_type=c(1, 3),
              id_v=c("ids", "second_ids"),
              excl_col=c(), rtn_col=c()))
   ids val ids bool second_ids val ids last second_ids val ids last
            a TRUE 13 <NA> <NA> <NA>
#1 a13
                                                     1
                                                         a oui
                          12 2 z non
8 <NA> <NA> <NA>
                                                12 2 z non
         7
#2 z12
             z FALSE
        2 z FALSE
                                               <NA> <NA> <NA> <NA>
#3
   z8
        4 a FALSE
                          34 <NA> <NA> <NA>
                                               <NA> <NA> <NA> <NA>
#4 a34
#5 a22
        1 a TRUE
                          22 <NA> <NA> <NA>
                                               <NA> <NA> <NA> <NA>
#6 a12
        2 a TRUE
                          12 <NA> <NA> <NA>
                                               <NA> <NA> <NA> <NA>
#7 a13 <NA> <NA> <NA>
                        <NA> <NA> <NA> <NA>
                                              <NA> <NA> <NA> <NA>
                        <NA> 1 a oui
<NA> <NA> <NA> <NA>
#8 all <NA> <NA> <NA>
                                                11 9 a oui
#9 z12 <NA> <NA> <NA>
                                               <NA> <NA> <NA> <NA>
```

10 appndr

#10 a8 <NA> <NA> <NA> 4 a oui

```
#
  second_ids
#1
         13
#2
         12
#3
       <NA>
       <NA>
#4
#5
       <NA>
#6
       <NA>
#7
       <NA>
#8
        11
#9
       <NA>
#10
          8
print(any_join_datf(inpt_datf_l=list(datf1, datf2, datf3), join_type=c(1), id_v=c("ids"),
             excl_col=c(), rtn_col=c()))
#ids val ids last second_ids val ids bool second_ids val ids last
11 3 a TRUE
12 7 z FALSE
8 4 a FALSE
                                          13 1 a oui
#2 a
      1 a oui
                     12
                                                2 z non
9 a oui
                                            12
#3 z
      2 z non
#4 a
      4 a oui
                                            34
# second_ids
#1
      <NA>
#2
        13
        12
#3
#4
        11
```

8 4

a oui

appndr

appndr

Description

Append to a vector "inpt_v" a special value "val" n times "mmn". The appending begins at "strt" index.

Usage

```
appndr(inpt_v, val = NA, hmn, strt = "max")
```

Arguments

inpt_v	is the input vector
val	is the special value
hmn	is the number of special value element added
strt	is the index from which appending begins, defaults to max which means the end of "inpt_v"

arroundr_mean 11

Examples

```
print(appndr(inpt_v=c(1:3), val="oui", hmn=5))
#[1] "1"    "2"    "3"    "oui" "oui" "oui" "oui" "oui"
print(appndr(inpt_v=c(1:3), val="oui", hmn=5, strt=1))
#[1] "1"    "oui" "oui" "oui" "oui" "oui" "2"    "3"
```

arroundr_mean

arroundr_mean

Description

Takes an ascendly int ordered vector as input and assigns each elements that are close enough to the same value accrdng to a step value (step_value), see examples.

Usage

```
arroundr_mean(inpt_v = c(), step_val)
```

Arguments

```
inpt_v is the input vector
step_val is the step_value
```

Examples

arroundr_min

arroundr_min

Description

Takes an ascendly int ordered vector as input and assigns each elements that are close enough to the same value accrdng to a step value (step_value), see examples.

Usage

```
arroundr_min(inpt_v = c(), step_val)
```

12 better_match

Arguments

```
inpt_v is the input vector
step_val is the step value
```

Examples

```
print(arroundr_min(inpt_v = c(-11:25), step_val = 5))

[1] -11 -11 -11 -11 -11 -11 -6 -6 -6 -6 -6 -1 -1 -1 -1 -1 4 4 4
[20] 4 4 9 9 9 9 9 14 14 14 14 14 19 19 19 19 19 24
```

better_match

better_match

Description

Allow to get the nth element matched in a vector

Usage

```
better_match(inpt_v = c(), ptrn, untl = 1, nvr_here = NA)
```

Arguments

inpt_v is the input vector

ptrn is the pattern to be matched

untl is the maximum number of matched pattern outputed

nvr_here is a value you are sure is not present in inpt_v

```
print(better_match(inpt_v=c(1:12, 3, 4, 33, 3), ptrn=3, untl=1))
#[1] 3
print(better_match(inpt_v=c(1:12, 3, 4, 33, 3), ptrn=3, untl=5))
#[1] 3 13 16
print(better_match(inpt_v=c(1:12, 3, 4, 33, 3), ptrn=c(3, 4), untl=5))
[1] 3 13 16 4 14
print(better_match(inpt_v=c(1:12, 3, 4, 33, 3), ptrn=c(3, 4), untl=c(1, 5)))
[1] 3 4 14
```

better_split 13

better_split better_split

Description

Allows to split a string by multiple split, returns a vector and not a list.

Usage

```
better_split(inpt, split_v = c())
```

Arguments

```
inpt is the input character
split_v is the vector containing the splits
```

Examples

```
print(better_split(inpt = "o-u_i", split_v = c("-")))
[1] "o" "u_i"
print(better_split(inpt = "o-u_i", split_v = c("-", "_")))
[1] "o" "u" "i"
```

```
better_split_any better_split_any
```

Description

Allows to split a string by multiple split regardless of their length, returns a vector and not a list. Contrary to better_split, this functions keep the delimiters in the output.

Usage

```
better_split_any(inpt, split_v = c())
```

Arguments

```
\begin{array}{ll} \text{inpt} & \text{is the input character} \\ \text{split\_v} & \text{is the vector containing the splits} \end{array}
```

14 better_sub

Examples

```
print(better_split_any(inpt = "o-u_i", split_v = c("-")))
[1] "o" "-" "u_i"
print(better_split_any(inpt = "o-u_i", split_v = c("-", "_")))
[1] "o" "-" "u" "_" "i"
[1] "--"
                "/"
          " 0 "
                                   "m"
                                                "m"
                                                       "/"
[10] " "
                "-opo-" "/"
                                          "-u"
          "_"
                             "m"
                                   "/"
                                                       "i-"
[19] "_"
          "__"
print(better_split_any(inpt = "(ok(ee:56))(ok2(oui)(ee:4))", split_v = c("(", ")", ":")))
        "ok" "("
[1] "("
                 "ee" ":"
                           "56" ")"
                                     ")" "(" "ok2" "(" "oui"
        "("
             "ee" ":"
                       "4"
                            ")"
                                ")"
 [13] ")"
```

better_sub

better_sub

Description

Allow to perform a sub operation to a given number of matched patterns, see examples

Usage

```
better_sub(inpt_v = c(), pattern, replacement, untl_v = c())
```

Arguments

inpt_v is a vector containing all the elements that contains expressions to be substituted
pattern is the expression that will be substituted
replacement is the expression that will substituate pattern
untl_v is a vector containing, for each element of inpt_v, the number of pattern that will be substituted

better_sub_mult 15

better_sub_mult

better_sub_mult

Description

Allow to perform a sub_mult operation to a given number of matched patterns, see examples

Usage

```
better_sub_mult(
  inpt_v = c(),
  pattern_v = c(),
  replacement_v = c(),
  untl_v = c()
)
```

Arguments

inpt_v is a vector containing all the elements that contains expressions to be substituted
pattern_v is a vector containing all the patterns to be substituted in any elements of inpt_v
replacement_v
 is a vector containing the expression that are going to substituate those provided
 by pattern_v

untl_v is a vector containing, for each element of inpt_v, the number of pattern that will
be substituted

16 better_unique

better_unique

better_unique

Description

Returns the element that are not unique from the input vector

Usage

```
better_unique(inpt_v, occu = ">-1-")
```

Arguments

inpt_v

is the input vector containing the elements

occu

is a parameter that specifies the occurence of the elements that must be returned, defaults to ">-1-" it means that the function will return all the elements that are present more than one time in inpt_v. The synthax is the following "comparaison_type-actual_value-". The comparaison type may be "==" or ">" or "<". Occu can also be a vector containing all the occurence that must have the elements to be returned.

```
print(better_unique(inpt_v=c("oui", "oui", "non", "non", "peut", "peut1", "non")))
#[1] "oui" "non"

print(better_unique(inpt_v=c("oui", "oui", "non", "non", "peut", "peut1", "non"), occu="=
#[1] "oui"

print(better_unique(inpt_v=c("oui", "oui", "non", "non", "peut", "peut1", "non"), occu=">
#[1] "non"

print(better_unique(inpt_v=c("oui", "oui", "non", "non", "peut", "peut1", "non"), occu=c("oui", "oui", "non", "peut", "peut1", "non")
```

bind_cols 17

```
print(better_unique(inpt_v = c("a", "b", "c", "c"), occu = "==-1-"))
[1] "a" "b"
print(better_unique(inpt_v = c("a", "b", "c", "c"), occu = "<-2-"))
[1] "a" "b"</pre>
```

bind_cols

bind_cols

Description

Allow to find the cols of a dataframe in an other dataframe, see examples

Usage

```
bind_cols(from_datf, in_datf)
```

Arguments

from_datf is the dataframe that contains the cols to find among other cols in_datf is the dataframe that only contans the cols to find in from_datf

```
iris[, 5] <- as.character(iris[, 5])</pre>
iris <- cbind(iris, iris[, 4])</pre>
from_datf <- iris</pre>
in_datf <- iris[, c(1, 2, 2, 2, 4)]</pre>
bind_cols(from_datf = from_datf,
           in_datf = in_datf)
[[1]]
[1] 1
[[2]]
[1] 2
[[3]]
[1] 2
[[4]]
[1] 2
[[5]]
[1] 4 6
```

18 can_be_num

bind_rows

bind_rows

Description

Allow to find the rows of a dataframe in an other dataframe, see examples

Usage

```
bind_rows(from_datf, in_datf)
```

Arguments

from_datf is the dataframe that contains the rows to find among other rows in_datf is the dataframe that only contans the rows to find in from_datf

Examples

can_be_num

can_be_num

Description

Return TRUE if a variable can be converted to a number and FALSE if not (supports float)

Usage

```
can_be_num(x)
```

closer_ptrn 19

Arguments

Х

is the input value

Examples

```
print(can_be_num("34.677"))
#[1] TRUE
print(can_be_num("34"))
#[1] TRUE
print(can_be_num("3rt4"))
#[1] FALSE
print(can_be_num(34))
#[1] TRUE
```

closer_ptrn

closer_ptrn

Description

Take a vector of patterns as input and output each chosen word with their closest patterns from chosen patterns.

Usage

```
closer_ptrn(
   inpt_v,
   base_v = c("?", letters),
   excl_v = c(),
   rtn_v = c(),
   sub_excl_v = c(),
   sub_rtn_v = c()
)
```

Arguments

inpt_v is the input vector containing all the patterns

must contain all the characters that the patterns are succeptible to contain, defaults to c("?", letters). "?" is necessary because it is internally the default value added to each element that does not have a sufficient length compared to the longest pattern in inpt_v. If set to NA, the function will find by itself the elements to be filled with but it may takes an extra time

excl_v

is the vector containing all the patterns from inpt_v to exclude for comparing them to others patterns. If this parameter is filled, so "rtn_v" must be empty.

20 closer_ptrn

is the vector containing all the patterns from inpt_v to keep for comparing them to others patterns. If this parameter is filled, so "rtn_v" must be empty.

sub_excl_v is the vector containing all the patterns from inpt_v to exclude for using them to compare to another pattern. If this parameter is filled, so "sub_rtn_v" must be empty.

sub_rtn_v is the vector containing all the patterns from inpt_v to retain for using them to compare to another pattern. If this parameter is filled, so "sub_excl_v" must be empty.

Examples

#[[15]]

```
print(closer_ptrn(inpt_v=c("bonjour", "lpoerc", "nonnour", "bonnour", "nonjour", "aurevoi
#[[1]]
#[1] "bonjour"
#[[2]]
#[1] "lpoerc"
                "nonnour" "bonnour" "nonjour" "aurevoir"
#[[3]]
#[1] 1 1 2 7 8
#[[4]]
#[1] "lpoerc"
#[[5]]
#[1] "bonjour" "nonnour" "bonnour" "nonjour" "aurevoir"
#[[6]]
#[1] 7 7 7 7 7
#[[7]]
#[1] "nonnour"
#[[8]]
#[1] "bonjour" "lpoerc"
                           "bonnour" "nonjour" "aurevoir"
#[[9]]
#[1] 1 1 2 7 8
#[[10]]
#[1] "bonnour"
#[[11]]
#[1] "bonjour" "lpoerc"
                           "nonnour" "nonjour"
                                                 "aurevoir"
#[[12]]
#[1] 1 1 2 7 8
#[[13]]
#[1] "nonjour"
#[[14]]
#[1] "bonjour" "lpoerc"
                           "nonnour" "bonnour"
                                                 "aurevoir"
```

closer_ptrn_adv 21

```
#[1] 1 1 2 7 8
#[[16]]
#[1] "aurevoir"
#[[17]]
#[1] "bonjour" "lpoerc" "nonnour" "bonnour" "nonjour"
#[[18]]
#[1] 7 8 8 8 8
print(closer_ptrn(inpt_v=c("bonjour", "lpoerc", "nonnour", "bonnour", "nonjour", "aurevoi
excl_v=c("nonnour", "nonjour"),
                sub_excl_v=c("nonnour")))
#[1] 3 5
#[[1]]
#[1] "bonjour"
#[[2]]
#[1] "lpoerc"
               "bonnour" "nonjour" "aurevoir"
#[[3]]
#[1] 1 1 7 8
#[[4]]
#[1] "lpoerc"
#[[5]]
#[1] "bonjour" "bonnour" "nonjour" "aurevoir"
#[[6]]#
#[1] 7 7 7 7
#[[7]]
#[1] "bonnour"
#[[8]]
#[1] "bonjour" "lpoerc" "bonnour" "nonjour" "aurevoir"
#[[9]]
#[1] 0 1 2 7 8
#[[10]]
#[1] "aurevoir"
#[[11]]
#[1] "bonjour" "lpoerc" "nonjour" "aurevoir"
#[[12]]
#[1] 0 7 8 8
```

22 closer_ptrn_adv

Description

Allow to find how patterns are far or near between each other relatively to a vector containing characters at each index ("base_v"). The function gets the sum of the indexes of each pattern letter relatively to the characters in base_v. So each pattern can be compared.

Usage

```
closer_ptrn_adv(
  inpt_v,
  res = "raw_stat",
  default_val = "?",
  base_v = c(default_val, letters),
  c_word = NA
)
```

Arguments

inpt_v	is the input vector containing all the patterns to be analyzed
res	is a parameter controling the result. If set to "raw_stat", each word in inpt_v will come with its score (indexes of its letters relatively to base_v). If set to something else, so "c_word" parameter must be filled.
default_val	is the value that will be added to all patterns that do not equal the length of the longest pattern in inpt_v. Those get this value added to make all patterns equal in length so they can be compared, defaults to "?"
base_v	is the vector from which all pattern get its result (letters indexes for each pattern relatively to base_v), defaults to c("default_val", letters). "default_val" is another parameter and letters is all the western alphabetic letters in a vector
c_word	is a pattern from which the nearest to the farest pattern in inpt_v will be compared

```
print(closer_ptrn_adv(inpt_v=c("aurevoir", "bonnour", "nonnour", "fin", "mois", "bonjour'
    res="word", c_word="bonjour"))

#[[1]]
#[1] 1 5 15 17 38 65
#
#[[2]]
#[1] "bonjour" "bonnour" "aurevoir" "nonnour" "mois" "fin"

print(closer_ptrn_adv(inpt_v=c("aurevoir", "bonnour", "nonnour", "fin", "mois")))

#[[1]]
#[1] 117 107 119 37 64
#
#[[2]]
#[1] "aurevoir" "bonnour" "nonnour" "fin" "mois"
```

clusterizer_v 23

```
clusterizer_v clusterizer_v
```

Description

Allow to output clusters of elements. Takes as input a vector "inpt_v" containing a sequence of number. Can also take another vector "w_v" that has the same size of inpt_v because its elements are related to it. The way the clusters are made is related to an accuracy value which is "c_val". It means that if the difference between the values associated to 2 elements is superior to c_val, these two elements are in distinct clusters. The second element of the outputed list is the begin and end value of each cluster.

Usage

```
clusterizer_v(inpt_v, w_v = NA, c_val)
```

Arguments

inpt_v is the vector containing the sequence of numberw_v is the vector containing the elements related to inpt_v, defaults to NAc_val is the accuracy of the clusterization

```
print(clusterizer_v(inpt_v=sample.int(20, 26, replace=TRUE), w_v=NA, c_val=0.9))
# [[1]]
#[[1]][[1]]
#[1] 1
#[[1]][[2]]
#[1] 2
#[[1]][[3]]
#[1] 3
#[[1]][[4]]
#[1] 4
#[[1]][[5]]
#[1] 5 5
#[[1]][[6]]
#[1] 6 6 6 6
#[[1]][[7]]
#[1] 7 7 7
#[[1]][[8]]
#[1] 8 8 8
#[[1]][[9]]
#[1] 9
```

24 clusterizer_v

```
#[[1]][[10]]
#[1] 10
#[[1]][[11]]
#[1] 12
#[[1]][[12]]
#[1] 13 13 13
#[[1]][[13]]
#[1] 18 18 18
#[[1]][[14]]
#[1] 20
#[[2]]
# [1] "1" "1" "-" "2" "2" "-" "3" "3" "-" "4" "4" "-" "5" "5" "-" #[16] "6" "6" "-" "7" "7" "-" "8" "8" "-" "9" "9" "-" "10" "10" "-"
#[31] "12" "12" "-" "13" "13" "-" "18" "18" "-" "20" "20"
print(clusterizer_v(inpt_v=sample.int(40, 26, replace=TRUE), w_v=letters, c_val=0.29))
#[[1]]
#[[1]][[1]]
#[1] "a"
#[[1]][[2]]
#[1] "b"
#[[1]][[3]]
#[1] "c" "d"
#[[1]][[4]]
#[1] "e" "f"
#[[1]][[5]]
#[1] "g" "h" "i" "j"
#[[1]][[6]]
#[1] "k"
#[[1]][[7]]
#[1] "1"
#[[1]][[8]]
#[1] "m" "n"
#[[1]][[9]]
#[1] "o"
#[[1]][[10]]
#[1] "p"
#[[1]][[11]]
```

#[1] "q" "r"

colins_datf 25

```
#[[1]][[12]]
#[1] "s" "t" "u"
#[[1]][[13]]
#[1] "v"
#[[1]][[14]]
#[1] "w"
#[[1]][[15]]
#[1] "x"
#[[1]][[16]]
#[1] "y"
#[[1]][[17]]
#[1] "z"
#[[2]]
# [1] "13" "13" "-" "14" "14" "-" "15" "15" "-" "16" "16" "-"
                                                                  "17" "17" "-"
#[16] "19" "19" "-" "21" "21" "-" "22" "22" "-" "23" "23" "-"
                                                                  "25" "25" "-"
#[31] "27" "27" "-" "29" "29" "-" "30" "30" "-" "31" "31" "-"
                                                                  "34" "34" "-"
#[46] "35" "35" "-" "37" "37"
```

colins_datf

colins_datf

Description

Allow to insert vectors into a dataframe.

Usage

```
colins_datf(inpt_datf, target_col = list(), target_pos = list())
```

Arguments

```
inpt_datf is the dataframe where vectors will be inserted
target_col is a list containing all the vectors to be inserted
target_pos is a list containing the vectors made of the columns names or numbers where
the associated vectors from target_col will be inserted after
```

26 col_convertr

```
# frst_col cur_col scd_col cur_col.1 cur_col
  1 oui 5 oui u
2 oui 4 oui z
#1
#2
                     3 2
        3
#3
             oui
                             oui
        4
                             non
#4
             non
                                      Z
                     1
#5
       5
                             non
             non
print(colins_datf(inpt_datf=datf1, target_col=list(c("oui", "oui", "oui", "non", "non"),
           c("u", "z", "z", "z", "u")),
              target_pos=list(c(1, 2), c("frst_col"))))
# frst_col cur_col scd_col cur_col cur_col
#1
      1 oui 5 u
                     4
       2
#2
            oui
                            Z
                                  oui
                     4 z oui
3 z oui
2 z non
1 u non
       3 oui
4 non
5 non
#3
#4
#5
```

```
col_convertr col_convertr
```

Description

Allow to convert all column that may be converted to a numeric, to a numeric, see examples

Usage

```
col_convertr(inpt_datf)
```

Arguments

```
inpt_datf is the input dataframe
```

```
datf <- mtcars
datf[, 3] <- as.character(datf[, 3])</pre>
datf[, 4] <- as.character(datf[, 4])</pre>
str(datf)
'data.frame': 32 obs. of 11 variables:
 $ mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num
              6 6 4 6 8 6 8 4 4 6 ...
              "160" "160" "108" "258" ...
 $ disp: chr
             "110" "110" "93" "110"
 $ hp : chr
 $ drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt : num 2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num 16.5 17 18.6 19.4 17 ...
 $ vs : num 0 0 1 1 0 1 0 1 1 1 ...
 $ am : num 1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num 4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num 4 4 1 1 2 1 4 2 2 4 ...
```

col_to_row 27

```
datf <- col_convertr(inpt_datf = datf)
all(datf == mtcars)
[1] TRUE</pre>
```

```
col_to_row
```

col_to_row

Description

Allow to reverse a dataframe (cols become rows and rows become cols)

Usage

```
col_to_row(inpt_datf)
```

Arguments

```
inpt_datf is the inout dataframe
```

Examples

```
datf_test <- data.frame(c(1:11), c(11:1))
print(col_to_row(inpt_datf = datf_test))

X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11
1 1 2 3 4 5 6 7 8 9 10 11
2 11 10 9 8 7 6 5 4 3 2 1</pre>
```

converter_date

converter_date

Description

Allow to convert any date like second/minute/hour/day/month/year to either second, minute...year. The input date should not necessarily have all its time units (second, minute...) but all the time units according to a format. Example: "snhdmy" is for second, hour, minute, day, month, year. And "mdy" is for month, day, year.

Usage

```
converter_date(inpt_date, convert_to, frmt = "snhdmy", sep_ = "-")
```

28 converter_format

Arguments

```
inpt_date is the input date
convert_to is the time unit the input date will be converted ("s", "n", "h", "d", "m", "y")
frmt is the format of the input date
sep_ is the separator of the input date. For example this input date "12-07-2012" has
"-" as a separator
```

Examples

```
print(converter_date(inpt_date="14-04-11-2024", sep_="-", frmt="hdmy", convert_to="m"))
#[1] 24299.15
print(converter_date(inpt_date="14-04-11-2024", sep_="-", frmt="hdmy", convert_to="y"))
#[1] 2024.929
print(converter_date(inpt_date="14-04-11-2024", sep_="-", frmt="hdmy", convert_to="s"))
#[1] 63900626400
print(converter_date(inpt_date="63900626400", sep_="-", frmt="s", convert_to="y"))
#[1] 2024.929
print(converter_date(inpt_date="2024", sep_="-", frmt="y", convert_to="s"))
#[1] 63873964800
```

```
converter_format converter_format
```

Description

Allow to convert a format to another

Usage

```
converter_format(inpt_val, sep_ = "-", inpt_frmt, frmt, default_val = "00")
```

Arguments

is the input value that is linked to the format

sep____ is the separator of the value in inpt_val

inpt_frmt is the format of the input value

frmt is the format you want to convert to

default_val is the default value given to the units that are not present in the input format

cost_and_taxes 29

Examples

cost_and_taxes

cost_and_taxes

Description

Allow to calculate basic variables related to cost and taxes from a bunch of products (elements). So put every variable you know in the following order:

Usage

```
cost_and_taxes(
   qte = NA,
   pu = NA,
   prix_ht = NA,
   tva = NA,
   prix_ttc = NA,
   prix_tva = NA,
   pu_ttc = NA,
   adjust = NA,
   prix_d_ht = NA,
   prix_d_ttc = NA,
   pu_d = NA,
   pu_d = NA,
   pu_d_ttc = NA
```

Arguments

```
is the quantity of elements
qte
                  is the price of a single elements without taxes
pu
                  is the duty-free price of the whole set of elements
prix_ht
                  is the percentage of all taxes
tva
                  is the price of all the elements with taxes
prix_ttc
prix_tva
                  is the cost of all the taxes
                  is the price of a single element taxes included
pu_ttc
                  is the discount percentage
adjust
                  is the free-duty price of an element after discount
prix_d_ht
                  is the price with taxes of an element after discount
prix_d_ttc
                  is the price of a single element after discount and without taxes
pu_d
                  is the free-duty price of a single element after discount
pu_d_ttc
```

30 cumulated_rows

Examples

```
print(cost_and_taxes(pu=45, prix_ttc=2111, qte=23))
# [1] 23.000000 45.000000 1.039614 2111.000000 1076.000000
# [7] 45.000000 NA NA NA NA NA
```

cumulated_rows

Description

Output a vector of size that equals to the rows number of the input dataframe, with TRUE value at the indices corresponding to the row where at least a cell of any column is equal to one of the values inputed in values_v

Usage

```
cumulated_rows(inpt_datf, values_v = c())
```

Arguments

inpt_datf is the input data.frame
values_v is a vector containing all the values that a cell has to equal to return a TRUE
value in the output vector at the index corresponding to the row of the cell

```
datf\_teste \leftarrow data.frame(c(1:10), c(10:1))
print(datf_teste)
   c.1.10. c.10.1.
1
        1
              10
2
         2
                 9
3
         3
                 8
4
         4
5
         5
                 6
                 5
6
         6
7
         7
                 4
8
                 3
         8
9
         9
                 2
10
        10
                 1
print(cumulated_rows(inpt_datf = datf_teste, values_v = c(2, 3)))
[1]
      FALSE TRUE TRUE FALSE
                                FALSE FALSE TRUE TRUE
                                                                   FALSE
```

cumulated_rows_na 31

Description

Output a vector of size that equals to the rows number of the input dataframe, with TRUE value at the indices corresponding to the row where at least a cell of any column is equal to NA.

Usage

```
cumulated_rows_na(inpt_datf)
```

Arguments

```
inpt_datf is the input data.frame
```

Examples

```
datf_teste <- data.frame(c(1, 2, 3, 4, 5, NA, 7), c(10, 9, 8, NA, 7, 6, NA))
print (datf_teste)
  c.1..2..3..4..5..NA..7. c.10..9..8..NA..7..6..NA.
1
                        1
2
                        2
                                                   9
3
                                                   8
                        3
4
                                                  NA
                        4
5
                        5
                                                   7
6
                       NA
                                                   6
                                                  NA
print(cumulated_rows_na(inpt_datf = datf_teste))
[1] FALSE FALSE FALSE TRUE FALSE TRUE TRUE
```

```
cutr_v cutr_v
```

Description

Allow to reduce all the elements in a vector to a defined size of nchar

Usage

```
cutr_v(inpt_v, untl = "min")
```

Arguments

```
inpt_v is the input vector
unt1 is the maximum size of nchar authorized by an element, defaults to "min", it
means the shortest element in the list
```

32 cut_v

Examples

```
test_v <- c("oui", "nonon", "ez", "aa", "a", "dsfsdsds")
print(cutr_v(inpt_v=test_v, untl="min"))
#[1] "o" "n" "e" "a" "a" "d"
print(cutr_v(inpt_v=test_v, untl=3))
#[1] "oui" "non" "ez" "aa" "a" "dsf"</pre>
```

cut_v

cut_v

Description

Allow to convert a vector to a dataframe according to a separator.

Usage

```
cut_v(inpt_v, sep_ = "")
```

Arguments

```
inpt_v is the input vector
sep_ is the separator of the elements in inpt_v, defaults to ""
```

data_gen 33

Description

Allo to generate in a csv all kind of data you can imagine according to what you provide

Usage

Arguments

type_	is a vector. Its arguments designates a column, a column can be made of numbers ("number"), string ("string") or both ("mixed")
strt_l	is a vector containing for each column the row from which the data will begin to be generated
nb_r	is a vector containing for each column, the number of row full from generated data
output	is the name of the output csv file, defaults to NA so no csv will be outputed by default
properties	is linked to type_distri because it is the parameters ("min_val-max_val") for "random type", ("u-x") for the poisson distribution, ("u-d") for gaussian distribution
type_distri	is a vector which, for each column, associate a type of distribution ("random", "poisson", "gaussian"), it meas that non only the number but also the length of the string will be randomly generated according to these distribution laws
str_source	is the source (vector) from which the character creating random string are (default set to the occidental alphabet)
round_l	is a vector which, for each column containing number, associate a round value, if the type of the value is numeric
sep_	is the separator used to write data in the csv

Value

new generated data in addition to saving it in the output

34 data_gen

```
print(data_gen())
# X1
      X2
            ХЗ
#1
  4
      2 <NA>
  2
      4
#2
           <NA>
#3 5 2
           <NA>
  2 abcd <NA>
#4
#5 4 abcd <NA>
#6 2 4
           <NA>
#7 2 abc <NA>
#8 4 abc <NA>
#9 4 3 <NA>
#10 4 abc abcd
#11 5 <NA> abc
#12 4 <NA>
           abc
#13 1 <NA>
           ab
#14 1 <NA> abcde
#15 2 <NA> abc
#16 4 <NA>
            а
#17 1 <NA> abcd
#18
   4 <NA>
           ab
#19 2 <NA> abcd
#20 3 <NA>
           ab
#21 3 <NA>
           abcd
#22 2 <NA>
#23 4 <NA>
            abc
#24 1 <NA> abcd
#25 4 <NA>
           abc
#26 4 <NA>
           ab
#27 2 <NA> abc
#28 5 <NA> ab
#29 3 <NA> abc
#30 5 <NA> abcd
#31 2 <NA> abc
#32 2 <NA> abc
#33 1 <NA>
           ab
#34 5 <NA>
            а
#35 4 <NA>
           ab
#36 1 <NA>
             ab
#37 1 <NA> abcde
#38 5 <NA> abc
#39
    4 <NA>
            ab
#40 5 <NA> abcde
#41 2 <NA>
#42
    3 <NA>
#43 2 <NA>
             ab
#44 4 <NA> abcd
#45 5 <NA>
           abcd
#46 3 <NA>
           abcd
#47 2 <NA>
           abcd
#48 3 <NA>
           abcd
#49 3 <NA> abcd
#50 4 <NA>
print(data_gen(strt_l=c(0, 0, 0), nb_r=c(5, 5, 5)))
```

data_meshup 35

```
# X1 X2 X3
#1 2 a abc
#2 3 abcde ab
#3 4 abcde a
#4 1 3 abc
#5 3 a abcd
```

data_meshup

data_meshup

Description

Allow to automatically arrange 1 dimensional data according to vector and parameters

Usage

```
data_meshup(
  data,
  cols = NA,
  file_ = NA,
  sep_ = ";",
  organisation = c(2, 1, 0),
  unic_sep1 = "_",
  unic_sep2 = "-"
)
```

Arguments

```
data
                     is the data provided (vector) each column is separated by a unic separator and
                     each dataset from the same column is separated by another unic separator (ex:
                     \mathtt{c}("",\,c("d",\,"\text{--"},\,"e",\,"\text{--"},\,"f"),\,\,"",\,\mathtt{c}("\mathtt{a}",\,"\mathtt{a}1",\,"\text{--"},\,"\mathtt{b}",\,"\text{--"},\,"\mathtt{c}",\,"\mathtt{c}1"),\,"\_")
cols
                     are the colnames of the data generated in a csv
                     is the file to which the data will be outputed, defaults to NA which means that
file_
                     the functio will return the dataframe generated and won't write it to a csv file
                     is the separator of the csv outputed
sep_
organisation is the way variables include themselves, for instance , resuming precedent ex-
                     ample, if organisation=c(1, 0) so the data output will be: d, a d, a1 e, c f, c f,
                     c1
                     is the unic separator between variables (default is "_")
unic_sep1
unic_sep2
                     is the unic separator between datasets (default is "-")
```

Examples

#3 e B

36 date_addr

```
#4 e r
#5 e uy
#6 f c
#7 f c1
```

date_addr

date_addr

Description

Allow to add or substract two dates that have the same time unit or not

Usage

```
date_addr(
  date1,
  date2,
  add = FALSE,
  frmt1,
  frmt2 = frmt1,
  sep_ = "-",
  convert_to = "dmy"
)
```

Arguments

```
date1 is the date from which the second date will be added or substracted
date2 is the date that will be added or will substract date1
add equals to FALSE if you want date1 - date2 and TRUE if you want date1 + date2
frmt1 is the format of date1 (snhdmy) (second, minute, hour, day, monthn year)
frmt2 is the format of date2 (snhdmy)
sep_ is the separator of date1 and date2
convert_to is the format of the outputed date
```

date_converter_reverse 37

Description

Allow to convert single date value like 2025.36 year to a date like second/minutehour/day/month/year (snhdmy)

Usage

```
date_converter_reverse(inpt_date, convert_to = "dmy", frmt = "y", sep_ = "-")
```

Arguments

inpt_date is the input date
convert_to is the date format the input date will be converted
frmt is the time unit of the input date
sep_ is the separator of the outputed date

date_converter_reverse

```
print(date_converter_reverse(inpt_date="2024.929", convert_to="hmy", frmt="y", sep_="-"))
#[1] "110-11-2024"

print(date_converter_reverse(inpt_date="2024.929", convert_to="dmy", frmt="y", sep_="-"))
#[1] "4-11-2024"

print(date_converter_reverse(inpt_date="2024.929", convert_to="hdmy", frmt="y", sep_="-")
#[1] "14-4-11-2024"

print(date_converter_reverse(inpt_date="2024.929", convert_to="dhym", frmt="y", sep_="-")
```

38 datf_appendr2

```
#[1] "4-14-2024-11"
```

datf_appendr

datf_appendr

Description

Allow to append all columns of a dataframe in a vector.

Usage

```
datf_appendr(inpt_datf)
```

Arguments

```
inpt_datf is the input dataframe
```

Examples

```
datf_teste <- data.frame("col1" = c(1:5), "col2" = c(5:1))
print(datf_appendr(inpt_datf = datf_teste))
[1] 1 2 3 4 5 5 4 3 2 1</pre>
```

datf_appendr2

datf_appendr2

Description

Allow to append all columns of a dataframe in a vector, specifying the column types ("integer" or "character"), see examples

Usage

```
datf_appendr2(inpt_datf, chs_type = "integer")
```

Arguments

```
inpt_datf is the inout dataframe
```

datf_insertr 39

Examples

```
datf_teste <- data.frame("col1" = c(1:5), "col2" = c(5:1),
    "col3" = c("oui", "oui", "oui", "non", "non"))

print(datf_appendr2(inpt_datf = datf_teste, chs_type = "integer"))

[1] 1 2 3 4 5 5 4 3 2 1

print(datf_appendr2(inpt_datf = datf_teste, chs_type = "character"))

[1] "oui" "oui" "oui" "non" "non"</pre>
```

datf_insertr

datf_insertr

Description

Insert rows after certain indexes, see examples

Usage

```
datf_insertr(inpt_datf, ids_vec, val_l)
```

Arguments

```
inpt_datf is the input dataframe
ids_vec is the ids where the rows has to be inserted after
val_l is a list containing all the rows (vector) to be inserted, linked to eevery index within ids_vec
```

Examples

```
datf \leftarrow data.frame(c(1:4), c(4:1))
print(datf)
  c.1.4. c.4.1.
1
     1
           4
2
       2
              3
3
       3
              2
       4
              1
print(datf_insertr(inpt_datf = datf, ids_vec = c(1, 3), val_l = list(c("non", "non"), c(")
  c.1.4. c.4.1.
1
       1
               4
2
      non
             non
21
       2
              3
3
       3
              2
5
      oui
             oui
        4
               1
```

print(datf_insertr(inpt_datf = datf, ids_vec = c(1, 3), val_l = list(c("non", "non"))))

40 datf_row_appendr2

```
datf_row_appendr datf_row_appendr
```

Description

Allow to append all rows of a dataframe in a vector.

Usage

```
datf_row_appendr(inpt_datf)
```

Arguments

```
inpt_datf is the input dataframe
```

Examples

```
datf_row_appendr2 datf_row_appendr2
```

Description

Allow to append all rows of a dataframe in a vector, specifying the column types ("integer" or "character"), see examples

Usage

```
datf_row_appendr2(inpt_datf, chs_type = "integer")
```

Arguments

```
inpt_datf is the inout dataframe
```

dcr_untl 41

Examples

```
datf_teste <- data.frame("col1" = c(1:5), "col2" = c(5:1),
    "col3" = c("oui", "oui", "oui", "non", "non"))

print(datf_row_appendr2(inpt_datf = datf_teste, chs_type = "integer"))

NULL

print(datf_row_appendr2(inpt_datf = datf_teste, chs_type = "character"))

col1 col2 col3 col1 col2 col3 col1 col2 col3 col1 col2 col3 col1
    "1" "5" "oui" "2" "4" "oui" "3" "3" "oui" "4" "2" "non" "5"
    col2 col3
    "1" "non"</pre>
```

dcr_untl

dcr_untl

Description

Allow to get the final value of a incremental or decremental loop.

Usage

```
dcr_untl(strt_val, cr_val, stop_val = 0)
```

Arguments

```
strt_val is the start value
cr_val is the incremental (or decremental value)
stop_val is the value where the loop has to stop
```

```
print(dcr_untl(strt_val=50, cr_val=-5, stop_val=5))
#[1] 9
print(dcr_untl(strt_val=50, cr_val=5, stop_val=450))
#[1] 80
```

42 delta_normal

dcr_val

 dcr_val

Description

Allow to get the end value after an incremental (or decremental loop)

Usage

```
dcr_val(strt_val, cr_val, stop_val = 0)
```

Arguments

```
strt_val is the start value
```

cr_val is the incremental or decremental value

stop_val is the value the loop has to stop

Examples

```
print(dcr_val(strt_val=50, cr_val=-5, stop_val=5))
#[1] 5
print(dcr_val(strt_val=47, cr_val=-5, stop_val=5))
#[1] 7
print(dcr_val(strt_val=50, cr_val=5, stop_val=450))
#[1] 450
print(dcr_val(strt_val=53, cr_val=5, stop_val=450))
#[1] 448
```

 $delta_normal$

delta_normal

Description

Returns the cumulative difference between a dataset of values and the normal distribution density formula, see examples

```
delta_normal(results_v = c(), mean_inpt, sd_inpt)
```

delta_unif 43

Arguments

results_v is the input vector

mean_inpt is the mea of the normal distribution

sd_inpt is the standard deviation of the normal distribution

Examples

```
x \leftarrow rnorm(n = 10000, mean = 15, sd = 1)
print(delta_normal(results_v = x,
                         mean\_inpt = 15,
                          sd_inpt = 3)
[1] 15904.85
x \leftarrow rnorm(n = 10000, mean = 15, sd = 2)
print(delta_normal(results_v = x,
                         mean\_inpt = 15,
                          sd_inpt = 3)
[1] 8050.662
x <- rnorm(n = 10000, mean = 15, sd = 5)
print(delta_normal(results_v = x,
                         mean\_inpt = 15,
                          sd_inpt = 3)
[1] 16219.87
x \leftarrow rnorm(n = 10000, mean = 15, sd = 4)
print(delta_normal(results_v = x,
                         mean\_inpt = 15,
                          sd_inpt = 3)
[1] 8081.28
x \leftarrow rnorm(n = 10000, mean = 15, sd = 3)
print(delta_normal(results_v = x,
                         mean\_inpt = 15,
                          sd_inpt = 3)
[1] 474.8151
```

delta_unif

delta_unif

Description

Returns the cumulative difference between a known uniform distribution and the input vector that may be a uniform distribution of given min and max, see examples.

```
delta_unif(inpt_v, min_inpt, max_inpt)
```

44 diff_datf

Arguments

inpt_v is the input vector that may represent a uniform distribution

min_inpt is the minimum of your uniform distribution
max_inpt is the maximum of your uniform distribution

Examples

```
print(delta_unif(inpt_v = runif(n = 5000, min = 12, max = 17), min_inpt = 12, max_inpt =
[1] 170.5542
print(delta_unif(inpt_v = runif(n = 5000, min = 12, max = 17), min_inpt = 122, max_inpt =
[1] 675102.8
```

```
depth_pairs_findr depth_pairs_findr
```

Description

Takes the pair vector as an input and associate to each pair a level of depth, see examples

Usage

```
depth_pairs_findr(inpt)
```

Arguments

inpt is the pair vector

Examples

```
print(depth_pairs_findr(c(1, 1, 2, 3, 3, 4, 4, 2, 5, 6, 7, 7, 6, 5)))
[1] 1 1 1 2 2 2 2 1 1 2 3 3 2 1
```

```
diff_datf
```

diff_datf

Description

Returns a vector with the coordinates of the cell that are not equal between 2 dataframes (row, column).

```
diff_datf(datf1, datf2)
```

dynamic_idx_convertr 45

Arguments

```
datf1 is an an input dataframe datf2 is an an input dataframe
```

Examples

```
datf1 <- data.frame(c(1:6), c("oui", "oui", "oui", "oui", "oui", "oui", "oui"), c(6:1))
datf2 <- data.frame(c(1:7), c("oui", "oui", "oui", "oui", "non", "oui", "zz"))
print(diff_datf(datf1=datf1, datf2=datf2))
#[1] 5 1 5 2</pre>
```

Description

Allow to convert the indices of vector ('from_v_ids') which are related to the each characters of a vector (from_v_val), to fit the newly established characters of the vector from_v_val, see examples.

Usage

```
dynamic_idx_convertr(from_v_ids, from_v_val)
```

Arguments

```
from_v_ids is the input vector of indices
from_v_val is the input vector of elements, or just the total number of characters of the elementsq in the vector
```

```
print(dynamic_idx_convertr(from_v_ids = c(1, 5), from_v_val = c("oui", "no", "ouI")))
[1] 1 2
print(dynamic_idx_convertr(from_v_ids = c(1, 6), from_v_val = c("oui", "no", "ouI")))
[1] 1 3
```

```
\verb|edm1_random_val| & edm1_random_val|
```

Description

Allow to generate a random number until a maximum length, see examples

Usage

```
edm1_random_val(len_untl = 5)
```

Arguments

len_untl is the maximum length that the random number could have

Examples

```
print(edm1_random_val(len_unt1 = 5))
[1] 54656
print(edm1_random_val(len_unt1 = 8))
[1] 64021015
print(edm1_random_val(len_unt1 = 3))
[1] 45
print(edm1_random_val(len_unt1 = 4))
[1] 6146
print(edm1_random_val(len_unt1 = 1))
```

Description

Allow to generate a random number of a pre determined length, see examples.

```
edm1_random_val_spe(len_inpt = 5)
```

edm1_rnorm1 47

Arguments

len_inpt i the lenght of the random number that will be generated

Examples

```
print (edm1_random_val_spe(len_inpt = 5))
[1] 55272
print (edm1_random_val_spe(len_inpt = 8))
[1] 79930782
print (edm1_random_val_spe(len_inpt = 3))
[1] 480
print (edm1_random_val_spe(len_inpt = 4))
[1] 6865
print (edm1_random_val_spe(len_inpt = 1))
[1] 2
```

 $\verb"edm1_rnorm1"$

edm1_rnorm1

Description

Reimplementation of rnorm function. You can also choose the most unlikely value to include in the outputed normal distribution. See examples. Warning, the lower sd_inpt is, the lower cur_step should be.

Usage

```
edm1_rnorm1(
  mean_inpt,
  sd_inpt,
  n_inpt,
  offset_proba = 1e-05,
  cur_step = "auto",
  accuracy_factor = 10
)
```

Arguments

```
mean_inpt is the mean of the normal distribution

sd_inpt is the standard deviation of the normal distribution

n_inpt is the number of values you want to generate

offset_proba is the value with the least probability to be included in the normal distribution
```

48 edm1_rnorm1

accuracy_factor

is an accuracy factor for the density of the values in the output vector (defaults to 10)

```
x \leftarrow edm1\_rnorm1 (mean\_inpt = 100,
                    sd_inpt = 15,
                    n_{inpt} = 15000,
                    offset_proba = 0.00001,
                    cur\_step = 0.3,
                    accuracy_factor = 10)
sd(x)
[1] 15.0456
summary(x)
  Min. 1st Qu. Median
                          Mean 3rd Qu.
                                           Max.
  43.13 89.93 100.43 100.30 110.33 159.53
x <- edm1_rnorm1 (mean_inpt = 100,</pre>
                    sd_inpt = 165,
                    n_{inpt} = 15000,
                    offset_proba = 0.00001,
                    cur\_step = 0.3,
                    accuracy_factor = 10)
sd(x)
[1] 164.1441
summary(x)
   Min. 1st Qu. Median Mean 3rd Qu.
                                           Max.
                 99.65 98.81 209.15 635.75
-444.55 -11.65
x \leftarrow edm1\_rnorm1 (mean\_inpt = 100,
                    sd_inpt = 0.45,
                    n_{inpt} = 15000,
                    offset_proba = 0.00001,
                    cur\_step = 0.05,
                    accuracy_factor = 10)
sd(x)
0.4504586
summary(x)
                          Mean 3rd Qu.
  Min. 1st Qu. Median
                                           Max.
  98.25 99.70 100.00
                         99.99 100.30 101.55
x <- edm1_rnorm1 (mean_inpt = 100,</pre>
                     sd_{inpt} = 15,
                      n_{inpt} = 15000,
```

edm1_rnorm2 49

```
offset_proba = 0.00001,
                      cur\_step = 0.05,
                      accuracy_factor = 10)
x <- sort(as.numeric(x))</pre>
library("ggplot2")
pdf("out.pdf")
length(x)
[1] 15000
sd(x)
[1] 15.11353
mean(x)
[1] 100.0294
datf <- data.frame("x" = c(1:length(x)),
                    "y" = x)
ggplot(data = datf, mapping = aes(x = x, y = y)) +
  geom_line() +
  theme_minimal()
x2 <- edm1_rnorm1 (mean_inpt = 100,</pre>
                      sd_{inpt} = 15,
                      n_{inpt} = 15000,
                      offset_proba = 0.00001,
                      cur\_step = 0.05,
                      accuracy_factor = 100)
x2 <- sort(as.numeric(x2))</pre>
sum(x == x2)
[1] 1742
```

edm1_rnorm2

 $edm1_rnorm2$

Description

edm1_rnorm1 but with an other algorithm that may be more accurate (especially for small sd_inpt) but needs more time to, see examples

```
edm1_rnorm2(
   mean_inpt,
```

50 edm1_rnorm2

```
sd_inpt,
n_inpt,
offset_proba = 1e-05,
cur_step = "auto",
accuracy_factor = 10
)
```

Arguments

mean_inpt is the mean of the normal distribution

sd_inpt is the standard deviation of the normal distribution
n_inpt is the number of values you want to generate

offset_proba is the value with the least probability to be included in the normal distribution accuracy_factor

is an accuracy factor for the density of the values in the output vector (defaults to 10)

```
x \leftarrow edm1\_rnorm2 (mean\_inpt = 100,
                       sd_inpt = 0.55,
                       n_{inpt} = 15000,
                       offset_proba = 0.00001,
                       cur\_step = 0.05,
                       accuracy_factor = 10)
x <- sort(as.numeric(x))</pre>
library("ggplot2")
pdf("out.pdf")
length(x)
[1] 15000
sd(x)
[1] 0.5512066
mean(x)
[1] 100.0044
datf <- data.frame("x" = c(1:length(x)),
                     "y" = x)
ggplot(data = datf, mapping = aes(x = x, y = y)) +
  geom_line() +
  theme_minimal()
x2 <- edm1_rnorm2(mean_inpt = 100,</pre>
                      sd_{inpt} = 0.55,
                       n_{inpt} = 15000,
                       offset_proba = 0.00001,
                       cur\_step = 0.05,
                       accuracy_factor = 100)
```

```
x2 <- sort(as.numeric(x2))
print(length(x2))
[1] 15000
sum(x == x2)
[1] 0</pre>
```

```
{\it edm1\_runif\_deterministic} \\ {\it edm1\_runif\_deterministic}
```

Description

Produces a deterministic uniform distribution, see examples

Usage

```
edm1_runif_deterministic(n_inpt, min_inpt, max_inpt)
```

Arguments

n_inpt is the number of wanted values

```
x <- edm1_runif_deterministic(n_inpt = 5000, min_inpt = 10, max_inpt = 15)
sd(x)
[1] 1.44352
sd(runif(n = 5000, min = 10, max = 15))
[1] 1.449532
x <- edm1_runif_deterministic(n_inpt = 5000, min_inpt = 10, max_inpt = 115)
sd(x)
[1] 30.31392
sd(runif(n = 5000, min = 10, max = 115))
[1] 30.33717</pre>
```

52 edm1_unif_time

```
edm1_unif_time edm1_unif_time
```

Description

Implementation of the runif function, see examples.

Usage

```
edm1_unif_time(
   n_inpt,
   min_inpt,
   max_inpt,
   random_seed = "random_data2.csv",
   divider_inpt = "auto"
)
```

Arguments

```
n_inpt is the number of valyues you want
min_inpt is the minimum in the uniform distribution
max_inpt is the maximum in the uniform distribution
random_seed is the csv of random values that the algo will use, defaults to R/random_data2.csv
divider_inpt is the prevalence of the random values (normally should not be changed, but you can play with it)
```

```
x \leftarrow edm1\_unif\_time(n\_inpt = 5000,
                      min_inpt = 10,
                      max_inpt = 15,
                      random_seed = "random_data2.csv",
                      divider_inpt = "auto")
################################### to compare to runif
library("ggplot2")
library("edm1")
pdf("out.pdf")
x \leftarrow edm1\_unif\_time(n\_inpt = 5000,
                      min_inpt = 10,
                      max_inpt = 15,
                      random_seed = "random_data2.csv",
                      divider_inpt = "auto")
x <- sort(as.numeric(x))</pre>
head(x)
xb \leftarrow round(x = x, digits = 1)
datf <- occu(xb)
#datf
```

edm_arrangr 53

```
ggplot(data = datf, mapping = aes(x = var, y = occurence)) +
   geom_col() +
   theme_minimal()

x <- runif(n = 5000, min = 10, max = 15)
x <- round(x = x, digits = 1)
datf <- occu(x)

ggplot(data = datf, mapping = aes(x = var, y = occurence)) +
   geom_col() +
   theme_minimal()</pre>
```

edm_arrangr

edm_arranger

Description

Arranges data according to the values of a variable, see examples

Usage

```
edm_arrangr(inpt_datf, col_order, top_n = 10, decreasing = TRUE)
```

Arguments

inpt_datf is the input dataframe

col_order is the column names or the column number of the variable that will be used to

arrange data

top_n is the top values

```
print(edm_arrangr(inpt_datf = mtcars, col_order = "wt", top_n = 15, decreasing = FALSE))
               mpg cyl disp hp drat
                                        wt qsec vs am gear carb
Lotus Europa
              30.4 4 95.1 113 3.77 1.513 16.90 1 1
                   4 75.7
                        75.7 52 4.93 1.615 18.52
71.1 65 4.22 1.835 19.90
Honda Civic
              30.4
                                                   1
                                                     1
                                                                2
Toyota Corolla 33.9
                     4
                                                   1
                                                      1
Fiat X1-9
              27.3
                     4 79.0
                              66 4.08 1.935 18.90
                                                   1
                    4 120.3
Porsche 914-2 26.0
                              91 4.43 2.140 16.70
                                                   0
                                                      1
```

```
Fiat 128 32.4 4 78.7 00 4.00 2.2.

Datsun 710 22.8 4 108.0 93 3.85 2.320 18.61
                                                1
                                                   1
                                                            1
                                                1
                                                   1
                                                            1
Toyota Corona 21.5 4 120.1 97 3.70 2.465 20.01
                                                1
                                                   0
                                                            1
             21.0 6 160.0 110 3.90 2.620 16.46
                                                0 1
Mazda RX4
Ferrari Dino 19.7 6 145.0 175 3.62 2.770 15.50 0 1
Volvo 142E
            21.4 4 121.0 109 4.11 2.780 18.60 1 1
Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1
Merc 230 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4
                                                           2
Ford Pantera L 15.8 8 351.0 264 4.22 3.170 14.50 0 1 5
Merc 240D
            24.4 4 146.7 62 3.69 3.190 20.00 1 0 4
```

54 edm_arrangr2

```
print(edm_arrangr(inpt_datf = mtcars, col_order = "wt", top_n = 10, decreasing = TRUE))
                    mpg cyl disp hp drat wt qsec vs am gear carb
Lincoln Continental 10.4 8 460.0 215 3.00 5.424 17.82
                                                       0
                                                         0
                        8 440.0 230 3.23 5.345 17.42
Chrysler Imperial 14.7
                                                      0 0
Cadillac Fleetwood 10.4 8 472.0 205 2.93 5.250 17.98
                                                      0 0
                   16.4 8 275.8 180 3.07 4.070 17.40
Merc 450SE
                                                      0 0
                                                              3
                                                                   3
Pontiac Firebird 19.2 8 400.0 175 3.08 3.845 17.05 0 0
                                                              3
                                                                   2
                  13.3 8 350.0 245 3.73 3.840 15.41 0 0
Camaro Z28
Merc 450SLC
                  15.2 8 275.8 180 3.07 3.780 18.00 0 0
Merc 450SL
                  17.3 8 275.8 180 3.07 3.730 17.60 0 0
                                                                   3
Duster 360
                  14.3 8 360.0 245 3.21 3.570 15.84 0 0
                                                             3
                                                                   4
Maserati Bora 15.0 8 301.0 335 3.54 3.570 14.60 0 1 Dodge Challenger 15.5 8 318.0 150 2.76 3.520 16.87 0 0
                                                             5
                                                                   8
                                                             3
                                                                   2.
```

edm_arrangr2

edm_arranger2

Description

Same as edm_arrangr but takes in count power like numbers for the values of the variable used to arrange the data.

Usage

```
edm_arrangr2(inpt_datf, col_order, top_n = 10, decreasing = TRUE)
```

Arguments

inpt_datf is the input dataframe

col_order is the column names or the column number of the variable that will be used to

arrange data

top_n is the top values

```
print(edm_arrangr2(inpt_datf = mtcars, col_order = "wt", top_n = 15, decreasing = FALSE))
```

```
mpg cyl disp hp drat
                                     wt qsec vs am gear carb
             30.4
                      95.1 113 3.77 1.513 16.90
Lotus Europa
                   4
                                              1
                                                1
Honda Civic
             30.4
                   4
                      75.7
                           52 4.93 1.615 18.52
                                              1
                                                 1
                     71.1
                           65 4.22 1.835 19.90
Toyota Corolla 33.9
                   4
                                              1
                                                 1
                                                     4
                                                          1
                  4 79.0
            27.3
                           66 4.08 1.935 18.90
Fiat X1-9
                                              1
                                                 1
                                                          1
Porsche 914-2 26.0 4 120.3 91 4.43 2.140 16.70
                                              0
                                                 1
            32.4 4 78.7
                           66 4.08 2.200 19.47
                                                1
Fiat 128
                                              1
                                                          1
            22.8 4 108.0 93 3.85 2.320 18.61
Datsun 710
                                                1
                                                         1
                                              1
Toyota Corona 21.5 4 120.1 97 3.70 2.465 20.01
                                              1 0
                                                         1
Mazda RX4
            21.0 6 160.0 110 3.90 2.620 16.46 0 1
Ferrari Dino 19.7 6 145.0 175 3.62 2.770 15.50 0 1 5
Volvo 142E 21.4 4 121.0 109 4.11 2.780 18.60 1 1 4
Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4
```

edm_group_by1 55

```
22.8 4 140.8 95 3.92 3.150 22.90 1 0
Ford Pantera L 15.8 8 351.0 264 4.22 3.170 14.50 0 1
                                                             5
Merc 240D 24.4 4 146.7 62 3.69 3.190 20.00 1 0
print(edm_arrangr2(inpt_datf = mtcars, col_order = "wt", top_n = 10, decreasing = TRUE))
                     mpg cyl disp hp drat wt qsec vs am gear carb
Lincoln Continental 10.4 8 460.0 215 3.00 5.424 17.82
                                                          0 0 3
Chrysler Imperial 14.7 8 440.0 230 3.23 5.345 17.42
                                                          0 0
                                                                        4
Cadillac Fleetwood 10.4 8 472.0 205 2.93 5.250 17.98 0 0 3
Merc 450SE 16.4 8 275.8 180 3.07 4.070 17.40 0 0 3
Pontiac Firebird 19.2 8 400.0 175 3.08 3.845 17.05 0 0 3
Camaro Z28 13.3 8 350.0 245 3.73 3.840 15.41 0 0 3 Merc 450SLC 15.2 8 275.8 180 3.07 3.780 18.00 0 0 3 Merc 450SL 17.3 8 275.8 180 3.07 3.730 17.60 0 0 3
                                                                       3
Merc 450SL
                  17.3 8 275.8 180 3.07 3.730 17.60 0 0 3
                                                                       3
Duster 360
                   14.3 8 360.0 245 3.21 3.570 15.84 0 0 3
                                                                       4
Maserati Bora 15.0 8 301.0 335 3.54 3.570 14.60 0 1 5 Dodge Challenger 15.5 8 318.0 150 2.76 3.520 16.87 0 0 3
                                                                       8
```

```
edm_group_by1
```

edm_group_by1

Description

Performs a group by (different algorythm than edm_group_by2), see examples

Usage

```
edm\_group\_by1(inpt\_datf, grp\_v = c())
```

Arguments

```
inpt_datf is the input dataframe
grp_v is the vector containiong the column names or the column numbers to perform
the group by, see examples
```

```
{\tt datf} \; \leftarrow \; {\tt data.frame} \, ("col1" \; = \; c \, ("A", \; "B", \; "B", \; "A", \; "C", \; "B") \, ,
                    "col2" = c("E", "R", "E", "E", "R", "R"),
                    "col3" = c("P", "P", "O", "O", "P", "O"))
print(datf)
  col1 col2 col3
    A E P
2
     В
          R
               P
        E
3
               0
     В
4
     A E O
5
     С
       R
              P
     B R
print(edm_group_by1(inpt_datf = datf, grp_v = c("col1")))
```

56 edm_group_by2

```
col1 col2 col3
  A E
       Ε
4
   Α
2
          P
      R
   В
   в Е
          0
3
   B R O
6
5
   C R
print(edm_group_by1(inpt_datf = datf, grp_v = c("col1", "col2")))
 col1 col2 col3
1
  A E
4
   A E
2
   B R
          Р
   B R
          0
6
3
  B E
          0
5
   C R
print(edm_group_by1(inpt_datf = datf, grp_v = c("col2", "col1", "col3")))
 col2 col1 col3
   Ε
      A
   E
       Α
           0
      В
          0
3
   E
      В
          P
2
   R
   R B
         0
6
   R C
print(edm_group_by1(inpt_datf = datf, grp_v = c("col2", "col1", "col3")))
 col2 col1 col3
   E A
   E A
3
  E B O
2 R B P
6 R B O
5
  R C P
```

```
edm_group_by2 edm_group_by2
```

Description

Performs a group by (different algorythm that edm_group_by1), see examples

Usage

```
edm_group_by2(inpt_datf, grp_v)
```

Arguments

```
inpt_datf is the input dataframe
```

edm_pert 57

grp_v

is the vector containiong the column names or the column numbers to perform the group by, see examples

Examples

```
datf <- data.frame("col1" = c("A", "B", "B", "A", "C", "B"),</pre>
                 "col2" = c("E", "R", "E", "E", "R", "R"),
                 "col3" = c("P", "P", "O", "O", "P", "O"))
print(datf)
 col1 col2 col3
1
    A E P
    В
       R
3
    В
       E
4
    A E
5
    C R
            P
        R
            0
    В
print(edm_group_by2(inpt_datf = datf, grp_v = c("col1")))
 col1 col2 col3
1
   A
       E
             Р
    Α
         Ε
              0
    В
         R
             Ρ
    В
         Ε
             0
6
    В
         R
             0
            P
5
    С
         R
print(edm_group_by2(inpt_datf = datf, grp_v = c("col1", "col2")))
 col1 col2 col3
1
   A E
             Р
         Ε
4
    Α
              0
3
    В
       E
2
    В
      R
           P
    В
      R
5
    С
         R
            P
print(edm_group_by2(inpt_datf = datf, grp_v = c("col2", "col1")))
 col2 col1 col3
1
    E A P
    Ε
              0
4
         Α
3
    Ε
         В
2
    R
         В
    R
         В
5
    R
         С
```

edm_pert

edm_pert

Description

Calculates margins and critical path of tasks based on PERT algorythm. The first tasks must be at the top of the input dataframe, see examples.

58 edm_pivot_longer1

Usage

```
edm_pert(inpt_datf)
```

Arguments

inpt_datf

is the input dataframe which contains all the tasks, their duration, their finish date at the earliest/latest and their antecedent, so the inpt_datf must contain 5 columns see examples

Examples

```
datf <- data.frame("task" = toupper(letters[1:7]),</pre>
                 "duration" = c(2, 8, 5, 2, 6, 5, 3),
                 "antecedent" = c(NA, NA, "A", "B", "B", "E", "A,D"),
                 "earliest" = c(2, 8, 19, 10, 14, 19, 19),
                 "latest" = c(14, 8, 19, 16, 14, 19, 19))
print(datf)
 task duration antecedent earliest latest
                         2
1
    A 2 <NA>
                                    14
2
            8
                    <NA>
                              8
    В
                                     8
                    А
3
    С
           5
                              19
                                     19
4
    D
           2
                      В
                              10
                                     16
5
            6
                              14
    Ε
                      В
                                     14
6
    F
            5
                      Ε
                              19
                                     19
7
    G
            3
                     A,D
                              19
                                     19
print(edm_pert(inpt_datf = datf))
[[1]]
 rtn_datf free_margin tot_margin
1
      Α
                 0
                         12
2
        В
                   0
                            0
3
       С
                 12
                           12
4
                  0
       D
                            6
5
       Ε
                  0
                            0
6
       F
                  0
                            0
7
                   6
[[2]]
[1] "B" "E" "F"
```

```
edm_pivot_longer1 edm_pivot_longer1
```

Description

Performs a pivot longer on dataframe, see examples. The synthax for variables must be value_id-modalitie_var1.modalitie_var2...

edm_pivot_longer1 59

Usage

```
edm_pivot_longer1(
  inpt_datf,
  col_vars = c(),
  col_vars_to = c(),
  null_value = c(0),
  nvr_here = "?"
)
```

Arguments

is the column name or the column number of the individuals

```
datf \leftarrow data.frame("individuals" = c(1, 2, 3),
                     c(1, 2, 3),
                     c(6, 0, 2),
                     c(7, 0, 0),
                     c(0, 0, 0),
                     c(1, 0, 4),
                     c(3, 0, 8),
                     c(9, 0 , 0),
                     c(11, 0, 5))
colnames(datf)[2:ncol(datf)] <- c("val1-A.R",</pre>
                                      "val1-A.T",
                                      "val1-B.R",
                                      "val1-B.T",
                                      "val2-A.R",
                                      "val2-A.T",
                                      "val2-B.R",
                                      "val2-B.T")
datf2 \leftarrow data.frame("individuals" = c(1, 2, 3),
                     c(7, 0, 2),
                     c(1, 0, 4),
                     c(9, 0, 8),
                     c(11, 22, 5))
colnames(datf2)[2:ncol(datf2)] <- c(</pre>
                           "val1-A",
                           "val1-B",
                           "val2-A",
                           "val2-B"
print(datf)
```

60 edm_pivot_longer1

```
individuals val1-A.R val1-A.T val1-B.R val1-B.T val2-A.R val2-A.T val2-B.R
      1
            1 6 7 0 1 3
2
                  2
          2
                         0
                                 0
                                         0
                                                 0
                                                        0
                 3
                         2
                                 0
                                        0
                                                        8
3
          3
                                                4
                                                                0
 val2-B.T
  11
1
2
       0
       5
3
print(edm_pivot_longer1(inpt_datf = datf,
                      col_vars = c(2:9),
                      individual_col = 1,
                      col_vars_to = c("Shape", "Way"),
                      null_value = c(0))
 individuals Shape Way val1 val2
  1 A R 1 1
1
                 Т
                          3
2
          1
                     6
               Α
                      7
                  R
3
          1
               В
                           9
                 T
                      0
4
          1
               В
                         11
                  R
          2
               Α
                       2
                     3
                 R
                           4
6
          3
               Α
                          8
7
          3
               Α
                  Τ
                       2
                 T
              В
                          5
8
          3
                      0
print(datf2)
 individuals val1-A val1-B val2-A val2-B
      1 7 1 9 11
                0
2
          2
                     0
                           0
                                 22
3
          3
                2
                      4
print(edm_pivot_longer1(inpt_datf = datf2,
                    col_vars = c(2:5),
                    individual_col = 1,
                    col_vars_to = c("Shape"),
                    null_value = c(0))
 individuals Shape val1 val2
       1 A 7 9
1
               В
                   1
                       11
2
          1
              B 0
A 2
B 4
3
          2
                       22
                      8
5
          3
5
          3
print(cur_data)
     individual country year twh_cons-biofuel_electricity
7475 France_1995 France 1995
                                              1.82
7503 France_2023 France 2023
                                              9.50
    twh_cons-coal_electricity twh_cons-gas_electricity
7475
                     24.18
                      2.16
                                           31.43
  twh_cons-hydro_electricity twh_cons-nuclear_electricity
                                              377.23
                      71.33
```

edm_pivot_longer2 61

```
7503
                         53.19
    twh_cons-oil_electricity twh_cons-other_renewable_exc_biofuel_electricity
7475
                       10.50
                                                                       0.51
                                                                       0.60
7503
                        9.71
    twh_cons-solar_electricity twh_cons-wind_electricity
7475
                                                   0.00
                         0.00
7503
                         23.26
                                                  48.61
print(edm_pivot_longer1(inpt_datf = cur_data,
                       col vars = c(4:ncol(cur data)),
                       col_vars_to = "type_energie"))
   individual country year
                                                     type_energie twh_cons
1 France_1995 France 1995
                                              biofuel_electricity 1.82
2 France_1995 France 1995
                                                coal_electricity 24.18
3 France_1995 France 1995
                                                  gas_electricity
                                                                     3.84
4 France_1995 France 1995
                                                hydro_electricity
                                                                    71.33
5 France_1995 France 1995
                                              nuclear_electricity 377.23
6 France_1995 France 1995
                                                  oil_electricity
                                                                     10.5
7
  France_1995 France 1995 other_renewable_exc_biofuel_electricity
                                                                    0.51
  France_2023 France 2023
                                              biofuel_electricity
                                                                      9.5
  France_2023 France 2023
                                                 coal_electricity
                                                                     2.16
                                                  gas_electricity 31.43
10 France_2023
               France 2023
11 France_2023 France 2023
                                                hydro_electricity
                                                                     53.19
                                                                  335.65
12 France_2023 France 2023
                                              nuclear_electricity
13 France_2023 France 2023
                                                  oil_electricity
                                                                    9.71
14 France_2023 France 2023 other_renewable_exc_biofuel_electricity
                                                                      0.6
15 France_2023 France 2023
                                               solar_electricity
                                                                   23.26
16 France_2023 France 2023
                                                 wind_electricity 48.61
```

```
edm_pivot_longer2 edm_pivot_longer2
```

Description

Performs a pivot longer on dataframe keeping the null values, see examples. The synthax for variables must be value_id-modalitie_var1.modalitie_var2...

Usage

```
edm_pivot_longer2(inpt_datf, col_vars = c(), col_vars_to = c())
```

Arguments

```
inpt_datf is the input dataframe
col_vars is a vector containing the column names or column numbers of the variables
col_vars_to is a vector containing the variables to which will be assign the modalities, see examples
individual_col
```

is the column name or the column number of the individuals

62 edm_pivot_longer2

```
datf \leftarrow data.frame("individuals" = c(1, 2, 3),
                 c(1, 2, 3),
                 c(6, 0, 2),
                 c(7, 0, 0),
                 c(0, 0, 0),
                 c(1, 0, 4),
                 c(3, 0, 8),
                 c(9, 0 , 0),
                 c(11, 0, 5))
colnames(datf)[2:ncol(datf)] <- c("val1-A.R",</pre>
                               "val1-A.T",
                               "val1-B.R",
                               "val1-B.T",
                               "val2-A.R",
                               "val2-A.T",
                               "val2-B.R",
                               "val2-B.T")
datf2 \leftarrow data.frame("individuals" = c(1, 2, 3),
                 c(7, 0, 2),
                 c(1, 0, 4),
                 c(9, 0, 8),
                 c(11, 22, 5))
colnames(datf2)[2:ncol(datf2)] <- c(</pre>
                      "val1-A",
                      "val1-B",
                      "val2-A",
                      "val2-B"
                 )
print(datf)
  individuals val1-A.R val1-A.T val1-B.R val1-B.T val2-A.R val2-A.T val2-B.R
  1 1 6 7 0 1 3 9
1
                                           0
                                                   0
                                                           0
                  2
                          0
                                  0
                                                                    0
2
           2
3
           3
                  3
                          2
                                   0
                                           0
                                                   4
                                                           8
                                                                    0
val2-B.T
  11
1
2
       0
        5
print(edm_pivot_longer2(inpt_datf = datf,
                       col_vars = c(2:9),
                        individual_col = 1,
                       col_vars_to = c("Shape", "Way")))
  individuals Shape Way val1 val2
          1 A R
                       1 1
1
2
           1
                 A T
3
           1
                B R
4
           1
                в т
                        0 11
5
           2
                A R 2 0
                        0 0
           2
                A T
```

edm_pivot_longer2 63

```
В
            2
                    R
8
            2
                 В
                     Τ
                         0
                    R
                         3
9
            3
                 Α
                              4
                    Т
                             8
10
            3
                          2
                 Α
                    R
                         0
11
            3
                 В
                              0
12
            3
                 В
                    Τ
                         0
                             5
print(datf2)
 individuals val1-A val1-B val2-A val2-B
           1 7 1 9 11
2
           2
                 0
                       0
                             0
                                    22
3
           3
                 2
                             8
                                    5
                       4
print(edm_pivot_longer2(inpt_datf = datf2,
                      col_vars = c(2:5),
                      individual_col = 1,
                      col_vars_to = c("Shape")))
 individuals Shape val1 val2
    1 A 7
               B 1 11
A 0 0
B 0 22
A 2 8
B 4 5
2
           1
                В
                     1
                         11
3
           2
4
           2
5
           3
           3
6
print(cur_data)
     individual country year twh_cons-biofuel_electricity
7475 France_1995 France 1995
7503 France_2023 France 2023
                                                  9.50
    twh_cons-coal_electricity twh_cons-gas_electricity
7475
                       24.18
                                               3.84
7503
                                               31.43
                        2.16
    twh_cons-hydro_electricity twh_cons-nuclear_electricity
7475
                                                 377.23
                        71.33
7503
                        53.19
                                                  335.65
    twh_cons-oil_electricity twh_cons-other_renewable_exc_biofuel_electricity
7475
                                                                     0.51
                      10.50
7503
                       9.71
                                                                     0.60
    twh_cons-solar_electricity twh_cons-wind_electricity
7475
                        0.00
                                                 0.00
7503
                        23.26
                                                 48.61
print(edm_pivot_longer2(inpt_datf = cur_data,
                      col_vars = c(4:ncol(cur_data)),
                      col_vars_to = "type_energie"))
   individual country
                                            vear
1 France_1995 France
                                            1995
2 France_1995 France
                                            1995
3 France_1995 France
                                            1995
4 France_1995 France
                                            1995
5 France_1995 France
                                            1995
```

64 edm_pivot_series

```
6 France_1995 France
                                              1995
7
  France_1995 France
                                              1995
8 France_1995 France
                                              1995
9 France_1995 France
                                              1995
10 France_2023 France
                                              2023
11 France_2023 France
                                              2023
12 France_2023 France
                                              2023
13 France_2023 France
                                              2023
14 France_2023 France
                                              2023
15 France 2023 France
                                              2023
16 France_2023 France
                                              2023
17 France_2023 France
                                              2023
18 France_2023 France
                                              2023
                            type_energie twh_cons
1
                      biofuel_electricity
                                             1.82
2
                        coal_electricity 24.18
3
                          gas_electricity
                                             3.84
4
                        hydro_electricity
                                            71.33
5
                      nuclear_electricity 377.23
                         oil_electricity
6
                                             10.5
  other_renewable_exc_biofuel_electricity
7
                                             0.51
8
                       solar_electricity
9
                         wind_electricity
                                                0
                                              9.5
10
                      biofuel_electricity
                        coal_electricity
                                             2.16
11
                                           31.43
12
                          gas_electricity
13
                        hydro_electricity
                                            53.19
                      nuclear_electricity
14
                                          335.65
                         oil_electricity
15
                                             9.71
16 other_renewable_exc_biofuel_electricity
                                              0.6
                                           23.26
17
                        solar electricity
18
                         wind_electricity
                                            48.61
```

```
edm_pivot_series edm_pivot_series
```

Description

Allow to create a new column for the value of the chosen columns at each new value of the column that represents the time. The occurence of each time stamp has to be equal, see examples (if not consider performing the time_serie_equalizer function fromm the same package)

Usage

```
edm_pivot_series(inpt_datf, time_col, col_v = NULL)
```

Arguments

inpt_datf	is the input dataframe
time_col	is the column name or number of the datafame
col_v	is a vector containing all the column numbers or names of the variables, if null all the column will be considered as variables apart from the column designated
	in time_col

edm_pivot_series 65

```
print(cur_datf)
  year
                               energy_source twh_cons
1 1995
                         biofuel_electricity 1.82
2 1995
                                              24.18
                           coal_electricity
3 1995
                                               3.84
                             gas_electricity
                                              71.33
4 1995
                          hydro_electricity
                          nuclear_electricity
5 1995
                                             377.23
6 1995
                             oil_electricity
                                              10.50
7 1995 other_renewable_exc_biofuel_electricity
                                               0.51
8 1995
                          solar_electricity
                                               0.00
                            wind_electricity
9 1995
                                               0.00
10 2023
                          biofuel_electricity
                                               9.50
11 2023
                           coal_electricity
                                               2.16
12 2023
                             gas_electricity 31.43
13 2023
                           hydro_electricity
                                              53.19
14 2023
                          nuclear_electricity 335.65
15 2023
                             oil_electricity 9.71
16 2023 other_renewable_exc_biofuel_electricity
                                               0.60
                           solar_electricity
17 2023
                                               23.26
18 2023
                            wind_electricity
                                             48.61
print(edm_pivot_series(inpt_datf = cur_datf, time_col = 1, col_v = c(3)))
                          energy_source twh_cons_1995 twh_cons_2023
1
                    biofuel_electricity 1.82 9.50
2
                                             24.18
                                                           2.16
                      coal_electricity
3
                        gas_electricity
                                              3.84
                                                          31.43
                                            71.33
                                                          53.19
4
                      hydro_electricity
5
                    nuclear_electricity
                                            377.23
                                                         335.65
                                            10.50
                       oil_electricity
                                                           9.71
7 other_renewable_exc_biofuel_electricity
                                              0.51
                                                           0.60
                                              0.00
                                                           23.26
                      solar_electricity
9
                       wind_electricity
                                              0.00
                                                           48.61
print(datf)
   individual year
                                          energy_source twh_cons
1 France_1995 1995
                                    biofuel_electricity 1.82
2 France_1995 1995
                                      coal_electricity 24.18
3 France_1995 1995
                                        gas_electricity
                                                          3.84
4 France_1995 1995
                                      hydro_electricity
                                                          71.33
                                                        377.23
5
  France_1995 1995
                                     nuclear_electricity
                                                        10.50
  France_1995 1995
                                        oil_electricity
7
  France_1995 1995 other_renewable_exc_biofuel_electricity
                                                           0.51
8 France_1995 1995
                                      solar_electricity
                                                           0.00
9 France_1995 1995
                                                           0.00
                                       wind_electricity
10 France_2023 2023
                                                          9.50
                                     biofuel_electricity
11 France_2023 2023
                                                          2.16
                                      coal_electricity
12 France_2023 2023
                                                        31.43
                                        gas_electricity
13 France_2023 2023
                                      hydro_electricity
                                                        53.19
14 France_2023 2023
                                    nuclear_electricity
                                                        335.65
15 France_2023 2023
                                        oil_electricity
                                                        9.71
16 France_2023 2023 other_renewable_exc_biofuel_electricity
                                                          0.60
17 France_2023 2023
                                      solar_electricity 23.26
```

66 edm_pivot_wider1

```
18 France_2023 2023
                                         wind_electricity 48.61
print(edm_pivot_series(inpt_datf = cur_datf, time_col = 2, col_v = c(1, 4)))
 individual_1995
                                         energy_source twh_cons_1995
    France_1995
1
                                    biofuel_electricity
                                                              1.82
2
     France_1995
                                      coal_electricity
                                                              24.18
3
    France_1995
                                       gas_electricity
                                                              3.84
4
    France_1995
                                     hydro_electricity
                                                              71.33
5
    France 1995
                                    nuclear_electricity
                                                             377.23
    France_1995
                                       oil_electricity
                                                             10.50
7
    France_1995 other_renewable_exc_biofuel_electricity
                                                              0.51
8
    France_1995
                                     solar_electricity
                                                              0.00
9
     France_1995
                                      wind_electricity
                                                               0.00
 individual_2023 twh_cons_2023
    France_2023 9.50
1
2
     France_2023
                         2.16
     France_2023
3
                       31.43
4
     France_2023
                        53.19
5
     France_2023
                       335.65
6
     France_2023
                         9.71
7
     France_2023
                         0.60
8
     France_2023
                        23.26
9
     France_2023
                        48.61
```

Description

Performs a pivot wider to a dataframe, see examples.

Usage

```
edm_pivot_wider1(inpt_datf, col_vars = c(), col_vals = c(), individual_col)
```

Arguments

```
inpt_datf is the input dataframe
col_vars is a vector containing the column names or column numbers of the variables to
    pivot

col_vals is a vector containing the column numbers or column names of the values to
    pivot
individual_col
```

is the column name or column number of the individuals

edm_pivot_wider2 67

```
\label{eq:data} \begin{array}{lll} \mbox{datf} <- \mbox{ data.frame("individual" = c(1, 1, 1, 2, 3, 3), \\ & \mbox{"var1" = c("A", "A", "B", "B", "B", "A"), \\ & \mbox{"var2" = c("R", "T", "T", "R", "T", "R"), \\ & \mbox{"val1" = c(6, 7, 1, 0, 4, 2), } \end{array}
                    "val2" = c(3, 9, 11, 22, 5, 8))
print(datf)
  individual var1 var2 val1 val2
          1 A R
                        6 3
2
           1
              A
                      Τ
           1 в
                    T 1 11
           2 B R 0 22
5
           3 в
                   T
                          4 5
                        2
                               8
           3 A
                   R
print(datf2)
  individual var1 val1 val2
        1 A 6 3
1
2
           1
                Α
                      7
3
           1
                В
                     1
                          11
                    0
4
           2
                В
                         22
5
           3
                В
                     4
                          5
                         8
                    2
           3
6
                Α
print(edm_pivot_wider1(
                        inpt_datf = datf,
                        col\_vars = c(2, 3),
                        col_vals = c(4, 5),
                        individual\_col = 1)
    )
  individuals val1-A.R val1-A.T val1-B.R val1-B.T val2-A.R val2-A.T val2-B.R
        1
                  6 7 0 1 3 9
                                                                             0
                                        0
                                                 0
                                                                    0
                                                                             22
2
            2
                     0
                               0
                                                           0
                             0
                                       0
                                                          8
3
            3
                    2
                                                 4
                                                                   0
                                                                             0
  val2-B.T
1 11
       0
2
print(edm_pivot_wider1(
                        inpt_datf = datf2,
                        col\_vars = c(2),
                        col_vals = c(3, 4),
                        individual\_col = 1)
    )
  individuals val1-A val1-B val2-A val2-B
            1 7 1 9 11
1
                   0
                                  0
                                         22
2
            2
                          0
3
                   2
                           4
                                 8
                                        5
```

68 edm_pivot_wider2

Description

Performs a pivot wider to a dataframe with a different algorythm than edm_pivot_wider, see examples.

Usage

```
edm pivot wider2(inpt datf, col vars = c(), col vals = c(), individual col)
```

Arguments

```
inpt_datf is the input dataframe
col_vars is a vector containing the column names or column numbers of the variables to
    pivot

col_vals is a vector containing the column numbers or column names of the values to
    pivot
individual_col
```

is the column name or column number of the individuals

```
datf2 \leftarrow data.frame("individual" = c(1, 1, 1, 2, 3, 3),
                  "var1" = c("A", "A", "B", "B", "B", "A"),
                  "val1" = c(6, 7, 1, 0, 4, 2),
                  "val2" = c(3, 9, 11, 22, 5, 8))
datf \leftarrow data.frame("individual" = c(1, 1, 1, 2, 3, 3),
                  "var1" = c("A", "A", "B", "B", "B", "A"),
                  "var2" = c("R", "T", "T", "R", "T", "R"),
                  "val1" = c(6, 7, 1, 0, 4, 2),
                  "val2" = c(3, 9, 11, 22, 5, 8))
print(datf)
  individual var1 var2 val1 val2
         1
             A
2
          1 A
                  Τ
                        7
                             9
3
          1 B T 1 11
          2 B R 0 22
4
5
          3 B T
                        4 5
6
          3
              A
                  R
                        2
                             8
print(datf2)
  individual var1 val1 val2
          1
              Α
2
          1
               Α
                    7
3
          1
              В
                   1
                        11
                  0
4
          2
              В
                        22
                 4
            В
                        5
5
          3
          3
                        8
6
               Α
print(edm_pivot_wider2(
                      inpt_datf = datf,
                      col\_vars = c(2, 3),
                      col_vals = c(4, 5),
                      individual\_col = 1)
   )
```

elements_equalifier 69

```
individuals val1-A.R val1-A.T val1-B.R val1-B.T val2-A.R val2-A.T val2-B.R
         1
                6 7 0 1 3
                                                      9
2
          2
                        0
                                0
                                        0
                                               0
                                                       0
                                                              22
                 0
                                0
3
          3
                 2
                        0
                                        4
                                               8
                                                      0
                                                              0
 val2-B.T
1
      11
2
       0
3
print(edm_pivot_wider2(
                   inpt_datf = datf2,
                   col\_vars = c(2),
                   col_vals = c(3, 4),
                   individual\_col = 1)
   )
 individuals val1-A val1-B val2-A val2-B
       1 7 1 9 11
1
                        0
                     0
2
          2
               0
                                22
3
          3
               2
                     4
```

```
elements_equalifier
```

elements_equalifier

Description

Takes an input vector with elements that have different occurence, and output a vector with all these elements with the same number of occurence, see examples

Usage

```
elements_equalifier(inpt_v, untl = 3)
```

Arguments

inpt_v is the input vector

untl is how many times each elements will be in the output vector

```
print (elements_equalifier(letters, untl = 2))

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z" "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
[39] "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

print (elements_equalifier(c(letters, letters[-1]), untl = 2))

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
[39] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z" "a"
```

70 extract_normal

```
equalizer_v equalizer_v
```

Description

Takes a vector of character as an input and returns a vector with the elements at the same size. The size can be chosen via depth parameter.

Usage

```
equalizer_v(inpt_v, depth = "max", default_val = "?")
```

Arguments

inpt_v is the input vector containing all the characters
depth is the depth parameter, defaults to "max" which means that it is equal to the character number of the element(s) in inpt_v that has the most
default_val is the default value that will be added to the output characters if those has an inferior length (characters) than the value of depth

Examples

```
print(equalizer_v(inpt_v=c("aa", "zzz", "q"), depth=2))
#[1] "aa" "zz" "q?"
print(equalizer_v(inpt_v=c("aa", "zzz", "q"), depth=12))
#[1] "aa?????????" "zzz???????" "q?????????"
```

extract_normal

extract_normal

Description

Allow to extract values that fits a normal distribution from any kind of dataset, see examples and parameters

```
extract_normal(
  inpt_datf,
  mean,
  sd,
  accuracy,
  round_value = 1,
  normalised = FALSE,
  n = NA,
  tries = 3
)
```

extract_normal 71

Arguments

inpt_datf is the input dataset as a dataframe, values/modalities are in the first column and frequency (not normalised) is in the second column is the mean of the target normal distribution mean sd is the standard deviation of the target normal distribution is how much of a difference beetween the points of the targeted normal distribuaccuracy tion and the actual points is tolerated is the round value for the normal distribution used under the hood to compare round_value the dataset and extract the best points, defaults to 1 is if the input frequency is divided by n, if TRUE the parameter n must be filled normalised n is the number of points is how many normal distributions are used under the hood to compare their tries points to the those in the input dataset, defaults to 3. The higher it is, the higher the number of different points from the input dataset will be in accordance for the normal distribution the function tries to build from the dataset. It does not increase by a lot but can be non-negligible and note that the higher the number of tries is, the higher the execution time of the function will be.

```
sample\_val \leftarrow round(rnorm(n = 72000, mean = 12, sd = 2), 1)
sample_freq <- unique_total(sample_val)</pre>
sample_qual <- infinite_char_seq(n = length(sample_freq))</pre>
datf_test <- data.frame(sample_qual, sample_freq)</pre>
n <- nrow(datf_test)</pre>
print(datf_test)
   sample_qual sample_freq
1
               а
2
               b
                          1155
3
                         1255
               С
4
               d
                          743
5
                е
                          696
               f
6
                          1028
7
                g
                          1160
8
                          1219
               h
9
               i
                          1353
10
                          1336
                j
11
               k
                          1308
12
               1
                           485
13
                          1306
               m
14
                          1429
                n
15
                           623
                0
16
                          1172
               р
17
                          1054
                q
18
                           999
               r
                           125
19
               S
2.0
                          1461
               t
21
                          1430
               u
22
               V
                           341
23
               W
                          1453
24
               Х
                           427
25
               У
                           869
```

72 extract_normal

26	Z	1395
27	aa	841
28	ab	952
29	ac	246
30	ad	468
31	ae	237
32	af	555
33	ag	1297
34		571
	ah	
35	ai	349
36	aj	773
37	ak	1086
38	al	1281
39	am	1471
40	an	1236
41	ao	394
42		1433
	ap	
43	aq	1328
44	ar	976
45	as	640
46	at	308
47	au	698
48	av	864
49	aw	1346
50	ax	1349
51	ay	6
52	az	1071
53	ba	248
54	bb	929
55	bc	925
56	bd	452
57	be	207
58	bf	546
59	bg	62
60	bh	107
61	bi	1184
62	bj	739
63	bk	624
64	bl	850
65	bm	1408
66	bn	620
67		
	bo	202
68	pp	10
69	pq	700
70	br	397
71	bs	1291
72	bt	178
73	bu	397
74	bv	1089
75	bw	1301
76	bx	328
77	by	1348
78	bz	97
79	ca	1452
80	cb	4
81	CC	100
82	cd	593
		5,5

extract_normal 73

83	ce	503
84	cf	164
85	cg	32
86	ch	259
87	ci	1089
88	сj	249
89	ck	165
90	cl	42
91	cm	143
92	cn	467
93	CO	347
94	ср	143
95	cq	69
96	cr	18
97	CS	290
98	ct	55
99	cu	141
100	CV	86
101	CW	303
102	CX	88
103	су	16
104	CZ	213
105	da	3
106	db	75
107	dc	32
108	dd	66
109	de	105
110	df	34
111	dg	56
112	dh	17
113	di	22
114	dj	120
115	dk	54
116	dl	9
117	dm	8
118	dn	36
119	do	20
120	dp	26
121	dq	54
121	dq dr	8
123	ds	10
123	dt	4
125	du	53
125	du dv	29
127	dv dw	1
127	dx	8
129		10
	dy	
130 131	dz	4 22
131	ea	
132	eb	9 17
133	ec	
	ed	55
135	ee	21
136	ef	6
137	eg	4
138	eh	3
139	ei	7

74 extract_normal

```
140
          еj
141
           ek
                        2
142
           el
143
           em
144
                        4
           en
145
                        1
           eo
146
           ер
                        2
          eq
                        3
147
148
          er
149
                        4
          es
150
                        3
          et
                        3
151
          eu
                        2
152
          ev
                       2
153
          ew
                       2
154
          ex
155
                        1
           ey
156
                        2
           ez
157
           fa
                        2
158
            fb
teste <- extract_normal(inpt_datf = datf_test,</pre>
                   mean = 10,
                    sd = 2,
                    accuracy = .1,
                    round_value = 1,
                    normalised = FALSE,
                    tries = 5)
print(length(unique(teste[, 1])) / n)
[1] 0.2848101 # so nearly 28.5 % of the different points were in
 #accordance with the construction of the target normal distribution
print(teste)
   values frequency
    dw 0.0001406866
1
       dw 0.0001406866
2
       dw 0.0001406866
3
4
      el 0.0002813731
5
       el 0.0002813731
6
       el 0.0002813731
7
       el 0.0002813731
8
       da 0.0004220597
9
       da 0.0004220597
10
       cb 0.0005627462
11
       cb 0.0005627462
12
       em 0.0007034328
      ay 0.0008441193
13
      ay 0.0008441193
14
15
      ei 0.0009848059
16
      ei 0.0009848059
17
      ei 0.0009848059
18
       dm 0.0011254924
```

19

20

21

bp 0.0014068655

cy 0.0022509848

cy 0.0022509848

extract_normal 75

```
22
        cy 0.0022509848
23
        dh 0.0023916714
24
        dh 0.0023916714
25
        cr 0.0025323579
26
        ee 0.0029544176
27
       di 0.0030951041
28
       dp 0.0036578503
       dp 0.0036578503
29
30
       cq 0.0045019696
31
       cq 0.0045019696
32
       df 0.0047833427
33
        dn 0.0050647158
34
       cl 0.0059088351
        cl 0.0059088351
35
        du 0.0074563872
36
        du 0.0074563872
37
38
        dg 0.0078784468
39
        dg 0.0078784468
       bg 0.0087225661
40
41
        bg 0.0087225661
42
        dd 0.0092853123
43
        cq 0.0097073720
44
        cq 0.0097073720
45
        a 0.0101294316
46
        cv 0.0120990433
        cx 0.0123804164
47
48
        cx 0.0123804164
        bz 0.0136465954
49
50
        cc 0.0140686550
        bh 0.0150534609
51
       bh 0.0150534609
52
53
        dj 0.0168823860
54
        s 0.0175858188
55
        s 0.0175858188
56
       cm 0.0201181767
57
       cf 0.0230725943
       ck 0.0232132808
58
59
       bt 0.0250422060
       bt 0.0250422060
60
61
       be 0.0291221159
62
       be 0.0291221159
63
       cz 0.0299662352
64
        cz 0.0299662352
65
        be 0.0291221159
66
        bo 0.0284186832
67
       bt 0.0250422060
       ck 0.0232132808
68
69
       ck 0.0232132808
70
        cm 0.0201181767
71
       cu 0.0198368036
72
        s 0.0175858188
73
       di 0.0168823860
74
       bh 0.0150534609
75
       bh 0.0150534609
76
        de 0.0147720878
77
       bz 0.0136465954
78
       bz 0.0136465954
```

76 extrt_only_v

```
79
        cx 0.0123804164
80
       cv 0.0120990433
81
       db 0.0105514913
82
       a 0.0101294316
83
       cq 0.0097073720
       dd 0.0092853123
84
85
       dd 0.0092853123
86
       bg 0.0087225661
87
       bg 0.0087225661
       dq 0.0078784468
88
89
       dk 0.0075970737
90
       du 0.0074563872
91
       cl 0.0059088351
       cl 0.0059088351
92
93
       dn 0.0050647158
94
       df 0.0047833427
95
       df 0.0047833427
       cg 0.0045019696
96
97
       dv 0.0040799100
       dp 0.0036578503
98
99
       di 0.0030951041
100
       di 0.0030951041
101
       ee 0.0029544176
102
        cr 0.0025323579
       dh 0.0023916714
103
       cy 0.0022509848
104
       cy 0.0022509848
105
       cy 0.0022509848
106
107
       cy 0.0022509848
       dl 0.0012661790
108
       dm 0.0011254924
109
110
       ei 0.0009848059
111
       ei 0.0009848059
112
       ay 0.0008441193
113
       ay 0.0008441193
114
       em 0.0007034328
       em 0.0007034328
115
116
       cb 0.0005627462
117
       cb 0.0005627462
118
       da 0.0004220597
119
       da 0.0004220597
120
       el 0.0002813731
121
       el 0.0002813731
122
       el 0.0002813731
123
       el 0.0002813731
124
       dw 0.0001406866
125
       dw 0.0001406866
126
       dw 0.0001406866
```

```
extrt_only_v extrt_only_v
```

Description

Returns the elements from a vector "inpt_v" that are in another vector "pttrn_v"

fillr 77

Usage

```
extrt_only_v(inpt_v, pttrn_v)
```

Arguments

```
inpt_v is the input vector
pttrn_v is the vector contining all the elements that can be in inpt_v
```

Examples

```
print(extrt_only_v(inpt_v=c("oui", "non", "peut", "oo", "ll", "oui", "non", "oui", "oui")
    pttrn_v=c("oui")))
#[1] "oui" "oui" "oui" "oui"
```

fillr fillr

Description

Allow to fill a vector by the last element n times

Usage

```
fillr(inpt_v, ptrn_fill = "\\.\\.\\d")
```

Arguments

inpt_v is the input vector

ptrn_fill is the pattern used to detect where the function has to fill the vector by the last element n times. It defaults to "...\d" where "\d" is the regex for an int value. So

this paramater has to have " \d " which designates n.

```
print(fillr(c("a", "b", "...3", "c")))
#[1] "a" "b" "b" "b" "c"
```

78 fold_rec

```
fixer_nest_v fixer_nest_v
```

Description

Retur the elements of a vector "wrk_v" (1) that corresponds to the pattern of elements in another vector "cur_v" (2) according to another vector "pttrn_v" (3) that contains the patter felements.

Usage

```
fixer_nest_v(cur_v, pttrn_v, wrk_v)
```

Arguments

```
cur_v is the input vector

pttrn_v is the vector containing all the patterns that may be contained in cur_v

wrk_v is a vector containing all the indexes of cur_v taken in count in the function
```

Examples

fold_rec

fold_rec

Description

Allow to get all the files recursively from a path according to an end and start depth value. If you want to have an other version of this function that uses a more sophisticated algorythm (which can be faster), check file_rec2. Depth example: if i have dir/dir2/dir3, dir/dir2b/dir3b, i have a depth equal to 3

Usage

```
fold_rec(xmax, xmin = 1, pathc = ".")
```

Arguments

xmax	is the end depth value
xmin	is the start depth value
pathc	is the reference path

fold_rec2 79

Description

Allow to find the directories and the subdirectories with a specified end and start depth value from a path. This function might be more powerfull than file_rec because it uses a custom algorythm that does not nee to perform a full recursive search before tuning it to only find the directories with a good value of depth. Depth example: if i have dir/dir2/dir3, dir/dir2b/dir3b, i have a depth equal to 3

Usage

```
fold_rec2(xmax, xmin = 1, pathc = ".")
```

Arguments

xmax	is the depth value
xmin	is the minimum value of depth
pathc	is the reference path, from which depth value is equal to 1

Description

Allow to convert xx-month-xxxx date type to xx-xx-xxxx

Usage

```
format_date(f_dialect, sentc, sep_in = "-", sep_out = "-")
```

Arguments

f_dialect are the months from the language of which the month come sentc is the date to convert sep_in is the separator of the dat input (default is "-") sep_out is the separator of the converted date (default is "-")

```
print(format_date(f_dialect=c("janvier", "février", "mars", "avril", "mai", "juin",
   "juillet", "aout", "septembre", "octobre", "novembre", "décembre"), sentc="11-septembre-2"
#[1] "11-09-2023"
```

geo_min

geo_min geo_min

Description

Return a dataframe containing the nearest geographical points (row) according to established geographical points (column).

Usage

```
geo_min(inpt_datf, established_datf)
```

Arguments

inpt_datf is the input dataframe of the set of geographical points to be classified, its firts column is for latitude, the second for the longitude and the third, if exists, is for the altitude. Each point is one row.

```
established_datf
```

is the dataframe containing the coordinates of the established geographical points

```
in_{-} \leftarrow data.frame(c(11, 33, 55), c(113, -143, 167))
in2_{-} \leftarrow data.frame(c(12, 55), c(115, 165))
print(geo_min(inpt_datf=in_, established_datf=in2_))
#
          X1
                    X2
   245.266
#1
                    NA
#2 24200.143
                    NA
#3
          NA 127.7004
in_{-} \leftarrow data.frame(c(51, 23, 55), c(113, -143, 167), c(6, 5, 1))
in2_ <- data.frame(c(12, 55), c(115, 165), c(2, 5))
print(geo_min(inpt_datf=in_, established_datf=in2_))
         Х1
                   X2
         NA 4343.720
#1
#2 26465.63
             NA
#3
        NA 5825.517
```

get_rec 81

Description

Allow to get the value of directorie depth from a path.

Usage

```
get_rec(pathc = ".")
```

Arguments

pathc is the reference path example: if i have dir/dir2/dir3, dir/dir2b/dir3b, i have a depth equal to 3

Description

Allow to calculate the distances between a set of geographical points and another established geographical point. If the altitude is not filled, so the result returned won't take in count the altitude.

Usage

```
globe(lat_f, long_f, alt_f = NA, lat_n, long_n, alt_n = NA)
```

Arguments

lat_f	is the latitude of the established geographical point
long_f	is the longitude of the established geographical point
alt_f	is the altitude of the established geographical point, defaults to NA
lat_n	is a vector containing the latitude of the set of points
long_n	is a vector containing the longitude of the set of points
alt_n	is a vector containing the altitude of the set of points, defaults to NA

82 grep_all

```
glue_groupr_v glue_groupr_v
```

Description

Takes an input vector and returns the same vector unlike that certain elements will be glued as an unique element according to thoses designated in a special vector, see examples.

Usage

```
glue_groupr_v(inpt_v, group_v = c(), untl)
```

Arguments

```
inpt_v is the input vector

a vector containing all the elements that will be glued in the output vector
```

Examples

```
print(glue_groupr_v(inpt_v = c("o", "-", "-", "u", "i", "-", "n",
    "o", "-", "-", "-", "zz", "/", "/"), group_v = c("-", "/")))

[1] "o" "--" "u" "i" "-" "n" "o" "---" "zz" "//"

print(glue_groupr_v(inpt_v = c("o", "-", "-", "u", "i", "-", "n",
    "o", "-", "-", "-", "-", "zz", "/", "/"), group_v = c("-", "/"), unt1 = 3))

[1] "o" "--" "u" "i" "-" "n" "o" "---" "-", "u", "i", "-", "n",
    "o", "-", "-", "-", "-", "zz", "/", "/"), group_v = c("-", "/"), unt1 = 2))

[1] "o" "--" "u" "i" "-" "n" "o" "---" "zz" "//"
```

```
grep_all grep_all
```

Description

Allow to perform a grep function on multiple input elements

Usage

```
grep_all(inpt_v, pattern_v)
```

Arguments

```
inpt_v is the input vectors to grep elements from
pattern_v is a vector containing the patterns to grep
```

grep_all2 83

Examples

grep_all2

grep_all2

Description

Performs the grep_all function with another algorythm, potentially faster

Usage

```
grep_all2(inpt_v, pattern_v)
```

Arguments

inpt_v is the input vectors to grep elements from
pattern_v is a vector containing the patterns to grep

84 groupr_datf

groupr_datf groupr_datf

Description

Allow to create groups from a dataframe. Indeed, you can create conditions that lead to a flag value for each cell of the input dataframeaccording to the cell value. This function is based on see_datf and nestr_datf2 functions.

Usage

```
groupr_datf(
  inpt_datf,
  condition_lst,
  val_lst,
  conjunction_lst,
  rtn_val_pos = c()
)
```

Arguments

```
interactive()
datf1 <- data.frame(c(1, 2, 1), c(45, 22, 88), c(44, 88, 33))
val_lst <- list(list(c(1), c(1)), list(c(2)), list(c(44, 88)))
condition_lst <- list(c(">", "<"), c("%%"), c("==", "=="))
conjunction_lst <- list(c("|"), c(), c("|"))
rtn_val_pos <- c("+", "++", "+++")
print(groupr_datf(inpt_datf=datf1, val_lst=val_lst, condition_lst=condition_lst, conjunction_lst=conjunction_lst, rtn_val_pos=rtn_val_pos))
# X1 X2 X3
#1 <NA> + +++
#2 ++ ++++
```

gsub_mult 85

```
#3 <NA> +++ +
```

gsub_mult gsub_mult

Description

Performs a gsub operation with n patterns and replacements.

Usage

```
gsub_mult(inpt_v, pattern_v = c(), replacement_v = c())
```

Arguments

is a vector containing the expression that are going to substituate those provided by pattern_v

Examples

```
historic_sequence1 historic_sequence1
```

Description

Allow to perform a pivot wider on a sequencial dataset (here the type is dataframe), each variable will be dupplicated in a column to show the value to this variable at n - 1 for each individual, see examples.

Usage

```
historic_sequence1(inpt_datf, bf_ = 1)
```

Arguments

```
inpt_datf is the input dataframe

bf_ is the number of previous value of the individual it will search for, see examples
```

86 historic_sequence2

```
set.seed(123)
var1 < - round(runif(n = 14, min = 100, max = 122))
set.seed(123)
var2 \leftarrow round(runif(n = 14, min = 14, max = 20))
datf <- data.frame("ids" = c(20, 20, 20, 20, 19, 19, 19, 18, 18, 18, 18,
                       17, 17, 17),
               "individual" = c("oui", "non", "peut1", "peut2", "oui", "peut1", "peut2"),
               "var1" = var1,
                "var2" = var2)
print(datf)
  ids individual var1 var2
  20 oui 106 16
           non 117
  20
2
                     19
         peut1 109 16
3
  20
         peut2 119 19
4
  20
           oui 121
5
  19
                     20
6 19
         peut1 101
                     14
7 19
         peut2 112
                    17
8 18
          oui 120 19
9 18
           non 112 17
10 18
         peut1 110
                    17
11 18
         peut2 121
                     20
12 17
          oui 110
                    17
13 17
          peutl 115
                     18
14 17
         peut2 113
                     17
historic_sequence1(inpt_datf = datf, bf_ = 2)
 id_seq individual var1-1 var1-2 var2-1 var2-2
        oui 121 120 20 19
1
    20
                                NA
2
                   NA
                         112
     20
             non
                                       17
                   101
                                14
3
     20
                         110
                                      17
           peut1
                                17
4
     20
                   112
                         121
                                      20
           peut2
5
                   120
                                19
     19
                         110
                                      17
            oui
                   110
                         115
                                17
                                      18
6
    19
           peut1
           peut2
    19
                  121
                         113
                                20
                                      17
historic_sequence1(inpt_datf = datf, bf_ = 3)
 id_seq individual var1-1 var1-2 var1-3 var2-1 var2-2 var2-3
1
    20
            oui 121 120 110 20 19 17
2
                   NA
                         112
                                      NA
                                            17
                                                  NA
     20
             non
                                NA
                                      14 17
17 20
3
                   101 110 115
    20
          peut1
                                                  18
4
    20
          peut2 112
                         121 113
                                                   17
```

historic_sequence2 87

Description

Allow to perform a pivot wider on a sequencial dataset (here the type is dataframe), each variable will be dupplicated in a column to show the value to this variable at n - 1 for each individual, see examples.

Usage

```
historic_sequence2(inpt_datf, bf_ = 1)
```

Arguments

```
is the input dataframe
inpt_datf
bf_
                  is the number of previous value of the individual it will search for, see examples
```

Examples

19

```
set.seed(123)
var1 < - round(runif(n = 14, min = 100, max = 122))
set.seed(123)
var2 \leftarrow round(runif(n = 14, min = 14, max = 20))
datf <- data.frame("ids" = c(20, 20, 20, 20, 19, 19, 19, 18, 18, 18, 18,
                            17, 17, 17),
                  "individual" = c("oui", "non", "peut1", "peut2",
                                    "oui", "peut1", "peut2"),
                  "var1" = var1,
                  "var2" = var2)
print(datf)
   ids individual var1 var2
1
   20 oui 106 16
2
   20
             non 117 19
          peut1 109 16
3
   20
          peut2 119 19
   20
5
  19
            oui 121
         peut1 101
peut2 112
6
  19
                         14
7
   19
                         17
            oui 120
8
   18
                         19
9
             non 112
                         17
   18
10 18
          peut1 110
                         17
11 18
          peut2 121
                         2.0
12
   17
             oui 110
                         17
13
   17
            peut1
                   115
                         18
   17
            peut2
                   113
                         17
print(historic_sequence2(inpt_datf = datf, bf_ = 2))
  id_seq individual var1-0 var1-1 var1-2 var2-0 var2-1 var2-2
1
              oui 106 121 120
                                           16 20
                                                          19
      20
2
      20
                       117
                                      112
                                              19
                                                             17
                               NA
                                                     NA
                non
      20
                      109
                            101
                                     110
                                                            17
3
                                             16
                                                     14
             peut1
              peut2 119 112 121
4
      20
                                             19
                                                    17
                                                            20

    oui
    121
    120
    110
    20
    19

    peut1
    101
    110
    115
    14
    17

    peut2
    112
    121
    113
    17
    20

5
      19
                                                            17
6
     19
            peut1
                                                            18
```

17

id_keepr

```
print(historic_sequence2(inpt_datf = datf, bf_ = 3))
 id_seq individual var1-0 var1-1 var1-2 var1-3 var2-0 var2-1 var2-2 var2-3
            oui 106 121 120 110 16 20 19 17
1
    2.0
                         NA 112
2
                  117
                                           19
                                                       17
     20
                                     NA
                                                 NA
             non
    20 peut1 109 101 110 115 16
20 peut2 119 112 121 113 19
3
                                                 14
                                                      17
                                                             18
                                                17
4
                                                      20
                                                             17
```

Description

Allow to get the original indexes after multiple equality comparaison according to the original number of row

Usage

```
id_keepr(inpt_datf, col_v = c(), el_v = c(), rstr_l = NA)
```

id_keepr

Arguments

inpt_datf	is the input dataframe
col_v	is the vector containing the column numbers or names to be compared to their respective elements in "el_v"
el_v	is a vector containing the elements that may be contained in their respective column described in " col_v "
rstr_l	is a list containing the vector composed of the indexes of the elements chosen for each comparison. If the length of the list is inferior to the length of comparisons, so the last vector of rstr_l will be the same as the last one to fill make rstr_l equal in term of length to col_v and el_v

incr_fillr 89

```
print(id_keepr(inpt_datf=datf1, col_v=c(1, 2), el_v=c("oui", "op"), rstr_l=list(c(1:5))))
#[1] 2 3
```

incr_fillr

incr_fillr

Description

Take a vector uniquely composed by double and sorted ascendingly, a step, another vector of elements whose length is equal to the length of the first vector, and a default value. If an element of the vector is not equal to its predecessor minus a user defined step, so these can be the output according to the parameters (see example):

Usage

```
incr_fillr(inpt_v, wrk_v = NA, default_val = NA, step = 1)
```

Arguments

inpt_v is the asending double only composed vector
wrk_v is the other vector (size equal to inpt_v), defaults to NA
default_val is the default value put when the difference between two following elements of inpt_v is greater than step, defaults to NA
step is the allowed difference between two elements of inpt_v

```
print(incr_fillr(inpt_v=c(1, 2, 4, 5, 9, 10),
                wrk_v=NA,
                default_val="increasing"))
#[1] 1 2 3 4 5 6 7 8 9 10
print(incr_fillr(inpt_v=c(1, 1, 2, 4, 5, 9),
               wrk_v=c("ok", "ok", "ok", "ok", "ok"),
                default_val=NA))
#[1] "ok" "ok" "ok" NA
                         "ok" "ok" NA
                                       NA
                                             NA
print(incr_fillr(inpt_v=c(1, 2, 4, 5, 9, 10),
                wrk_v=NA,
                default_val="NAN"))
                                   "NAN" "NAN" "NAN" "9"
#[1] "1"
           "2"
                "NAN" "4"
                             "5"
                                                           "10"
```

90 inner_all

```
infinite_char_seq infinite_char_seq
```

Description

Allow to generate an infinite sequence of unique letters

Usage

```
infinite_char_seq(n, base_char = letters)
```

Arguments

n is how many sequence of numbers will be generated

base_char is the vector containing the elements from which the sequence is generated

Examples

```
print(infinite_char_seq(28))

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
[16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "a" "aa" "ab"
```

```
inner_all inner_all
```

Description

Allow to apply inner join on n dataframes, datatables, tibble

Usage

```
inner_all(..., keep_val = FALSE, id_v)
```

Arguments

```
. . . are all the dataframes etc
```

keep_val is if you want to keep the id column

id_v is the common id of all the dataframes etc

insert_datf 91

Examples

```
datf1 <- data.frame(
    "id1"=c(1:5),
    "var1"=c("oui", "oui", "oui", "non", "non")
)

datf2 <- data.frame(
    "id1"=c(1, 2, 3, 7, 9),
    "var1"=c("oui2", "oui2", "oui2", "non2", "non2")
)

print(inner_all(datf1, datf2, keep_val=FALSE, id_v="id1"))

id1 var1.x var1.y
1 1 oui oui2
2 2 oui oui2
3 3 oui oui2</pre>
```

insert_datf

insert_datf

Description

Allow to insert dataframe into another dataframe according to coordinates (row, column) from the dataframe that will be inserted

Usage

```
insert_datf(datf_in, datf_ins, ins_loc)
```

Arguments

datf_in is the dataframe that will be inserted
datf_ins is the dataset to be inserted
ins_loc is a vector containg two parameters (row, column) of the begining for the insertion

```
datf1 \leftarrow data.frame(c(1, 4), c(5, 3))
datf2 \leftarrow data.frame(c(1, 3, 5, 6), c(1:4), c(5, 4, 5, "ereer"))
print(insert_datf(datf_in=datf2, datf_ins=datf1, ins_loc=c(4, 2)))
    c.1..3..5..6. c.1.4. c.5..4..5...ereer..
# 1
                      1
                1
# 2
                 3
                        2
                                              4
# 3
                 5
                        3
                        1
print(insert_datf(datf_in=datf2, datf_ins=datf1, ins_loc=c(3, 2)))
```

92 intersect_mod

```
c.1..3..5..6. c.1.4. c.5..4..5...ereer..
# 1
                 1
             1
# 2
              3
                    2
# 3
             5
                    1
# 4
              6
                    4
                                      3
print(insert_datf(datf_in=datf2, datf_ins=datf1, ins_loc=c(2, 2)))
 c.1..3..5..6. c.1.4. c.5..4..5...ereer..
# 1
# 2
             3
                                      5
                   1
# 3
                                      3
             5
                   4
            6 4
# 4
```

Description

Allows to calculate the intersection between n vectors

Usage

```
intersect_all(...)
```

Arguments

is all the vector you want to calculate the intersection from

Examples

```
print(intersect_all(c(1:5), c(1, 2, 3, 6), c(1:4)))
[1] 1 2 3
```

intersect_mod

intersect_mod

Description

Returns the mods that have elements in common

Usage

```
intersect_mod(datf, inter_col, mod_col, n_min, descendly_ordered = NA)
```

intersect_mod 93

Arguments

datf is the input dataframe is the column name or the column number of the values that may be commun inter col betwee the different mods mod col is the column name or the column number of the mods in the dataframe is the minimum elements in common a mod should have to be taken in count n_min ordered_descendly in case that the elements in commun are numeric, this option can be enabled by

giving a value of TRUE or FALSE see examples

Examples

non

4

```
datf <- data.frame("col1"=c("oui", "oui", "oui", "oui", "oui", "oui",</pre>
                     "non", "non", "non", "non", "ee", "ee", "ee"), "col2"=c(1:6, 2:5, 1:
print(intersect_mod(datf=datf, inter_col=2, mod_col=1, n_min=2))
  coll col2
2
   oui
           2
3
   oui
           3
7
   non
           2
8
   non
           3
12
           2
     ee
13
     ee
           3
print(intersect_mod(datf=datf, inter_col=2, mod_col=1, n_min=3))
  coll col2
2.
  oui
           2.
3
  oui
           3
4
           4
  oui
5
  oui
           5
7
           2
  non
8
           3
  non
9
   non
           4
10 non
           5
print(intersect_mod(datf=datf, inter_col=2, mod_col=1, n_min=5))
  col1 col2
1 oui
        1
2
  oui
          2
  oui
          3
  oui
          4
5
  oui
          5
6 oui
datf <- data.frame("col1"=c("non", "non", "oui", "oui", "oui", "oui",</pre>
                      "non", "non", "non", "ee", "ee", "ee"), "col2"=c(1:6, 2:5, 1
print(intersect_mod(datf=datf, inter_col=2, mod_col=1, n_min=3))
   col1 col2
   non
           3
```

94 intersect_mod2

```
10 non 5
3 oui 3
4 oui 4
5 oui 5
```

```
intersect_mod2 in
```

intersect_mod2

Description

Returns the mods that have elemnts in common, uses a different algorith than intersect_mod1, see examples

Usage

```
intersect_mod2(inpt_datf, inter_col, mod_col, n_min = 1)
```

Arguments

```
inpt_datf is the input dataframe
inter_col is the column number or name of the elements that may be in common to each
mod
mod_col is the column number or name of the mods
n_min is the minimum number of elements in common each mod have to have to be outputed in the returned dataframe
```

```
datf <- data.frame("country" = c("Germany", "France", "Italy", "France", "Germany", "Italy")</pre>
                   "year" = c(2000, 2000, 2001, 2002, 2003, 2004),
                   "random_value" = c(1, 3, 2, 1, 2, 2))
print(datf)
  country year random_value
1 Germany 2000
2 France 2000
   Italy 2001
                          2
4 France 2002
                          2
5 Germany 2003
   Italy 2004
print(intersect_mod2(inpt_datf = datf, inter_col = "year", mod_col = "country"))
  country year random_value
1 Germany 2000
2 France 2000
                          3
```

inter_max 95

Description

Takes as input a list of vectors composed of ints or floats ascendly ordered (intervals) that can have a different step to one of another element ex: list(c(0, 2, 4), c(0, 4), c(1, 2, 2.3)). The function will return the list of lists altered according to the maximum step found in the input list.

Usage

```
inter_max(inpt_l, max_ = -1000, get_lst = TRUE)
```

Arguments

```
inpt_l is the input list

max_ is a value you are sure is the minimum step value of all the sub-lists

get_lst is the parameter that, if set to True, will keep the last values of vectors in the return value if the last step exceeds the end value of the vector.
```

```
print(inter_max(inpt_l=list(c(0, 2, 4), c(0, 4), c(1, 2, 2.3)), get_lst=TRUE))

#[[1]]
#[1] 0 4
#
#[[2]]
#[1] 0 4
#
#[[3]]
#[1] 1.0 2.3

print(inter_max(inpt_l=list(c(0, 2, 4), c(0, 4), c(1, 2, 2.3)), get_lst=FALSE))

# [[1]]
#[1] 0 4
#
#[[2]]
#[1] 0 4
#
#[[3]]
#[1] 1
```

96 inter_min

inter_min

inter_min

Description

Takes as input a list of vectors composed of ints or floats ascendly ordered (intervals) that can have a different step to one of another element ex: list(c(0, 2, 4), c(0, 4), c(1, 2, 2.3)). This function will return the list of vectors with the same steps preserving the begin and end value of each interval. The way the algorythmn searches the common step of all the sub-lists is also given by the user as a parameter, see how_to paramaters.

Usage

```
inter_min(
  inpt_l,
  min_ = 1000,
  sensi = 3,
  sensi2 = 3,
  how_to_op = c("divide"),
  how_to_val = c(3)
)
```

Arguments

inpt_l	is the input list containing all the intervals
min_	is a value you are sure is superior to the maximum step value in all the intervals
sensi	is the decimal accuracy of how the difference between each value n to $n+1$ in an interval is calculated
sensi2	is the decimal accuracy of how the value with the common step is calculated in all the intervals
how_to_op	is a vector containing the operations to perform to the pre-common step value, defaults to only "divide". The operations can be "divide", "substract", "multiply" or "add". All type of operations can be in this parameter.
how_to_val	is a vector containing the value relatives to the operations in hot_to_op, defaults to 3 output from ex:

```
print(inter_min(inpt_l=list(c(0, 2, 4), c(0, 4), c(1, 2, 2.3))))
# [[1]]
# [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8
#[20] 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7
#[39] 3.8 3.9 4.0
#
#[[2]]
# [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8
#[20] 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7
#[39] 3.8 3.9 4.0
#
#[[3]]
```

isnt_divisible 97

```
# [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3
```

Description

Takes a vector as an input and returns all the elements that are not divisible by all choosen numbers from another vector.

Usage

```
isnt_divisible(inpt_v = c(), divisible_v = c())
```

Arguments

```
\begin{array}{ll} \text{inpt\_v} & \text{is the input vector} \\ \text{divisible\_v} & \text{is the vector containing all the numbers that will try to divide those contained in} \\ \text{inpt\_v} & \end{array}
```

Examples

```
print(isnt_divisible(inpt_v=c(1:111), divisible_v=c(2, 4, 5)))
# [1] 1 3 7 9 11 13 17 19 21 23 27 29 31 33 37 39 41 43 47
# [20] 49 51 53 57 59 61 63 67 69 71 73 77 79 81 83 87 89 91 93
# [39] 97 99 101 103 107 109 111
```

```
is_divisible is_divisible
```

Description

Takes a vector as an input and returns all the elements that are divisible by all choosen numbers from another vector.

Usage

```
is\_divisible(inpt\_v = c(), divisible\_v = c())
```

Arguments

```
\label{eq:containing} \begin{array}{ll} \text{inpt\_v} & \text{is the input vector} \\ \text{divisible\_v} & \text{is the vector containing all the numbers that will try to divide those contained in} \\ \text{inpt\_v} & \\ \end{array}
```

98 join_n_lvl

Examples

```
print(is_divisible(inpt_v=c(1:111), divisible_v=c(2, 4, 5)))
#[1] 20 40 60 80 100
```

join_n_lvl

join_n_lvl

Description

Allow to see the progress of the multi-level joins of the different variables modalities. Here, multi-level joins is a type of join that usually needs a concatenation of two or more variables to make a key. But here, there is no need to proceed to a concatenation. See examples.

Usage

```
join_n_lvl(frst_datf, scd_datf, join_type = c(), lst_pair = list())
```

Arguments

```
is the first data.frame (table)

scd_datf is the second data.frame (table)

join_type is a vector containing all the join type ("left", "inner", "right") for each variable is a lis of vectors. The vectors refers to a multi-level join. Each vector should have a length of 1. Each vector should have a name. Its name refers to the column name of multi-level variable and its value refers to the column name of the join variable.
```

Examples

|==| 100%

[1] "pair: vil2 idl2"

```
datf3 <- data.frame("vil"=c("one", "one", "one", "two", "two", "two"),</pre>
                     "charac"=c(1, 2, 2, 1, 2, 2),
                     "rev"=c(1250, 1430, 970, 1630, 2231, 1875),
                     "vil2" = c("one", "one", "one", "two", "two", "two"),
                     "id12" = c(1:6))
datf4 <- data.frame("vil"=c("one", "one", "one", "two", "two", "three"),</pre>
                    "charac"=c(1, 2, 2, 1, 1, 2),
                     "rev"=c(1.250, 1430, 970, 1630, 593, 456),
                     "vil2" = c("one", "one", "one", "two", "two"),
                     "idl2" = c(2, 3, 1, 5, 5, 5))
print(join_n_lvl(frst_datf=datf3, scd_datf=datf4, lst_pair=list(c("charac" = "vil"), c("v
                 join_type=c("inner", "left")))
[1] "pair: charac vil"
     0%
1
|= | 50%
2
```

just_anything 99

just_anything

just_anything

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_anything(inpt_v, symbol_ = "-", anything_v = c())
```

Arguments

```
inpt_v is the input vector
symbol_ is the chosen symbol to replace numbers
```

Examples

```
print(just_anything(inpt_v = c("oui222jj644", "oui122jj"),
symbol_ = "-", anything_v = letters))
[1] "oui-jj-" "oui-jj"
```

just_anything2

just_anything2

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_anything2(inpt_v, symbol_ = "-", anything_v = c())
```

just_chr

Arguments

```
inpt_v is the input vector
symbol_ is the chosen symbol to replace numbers
```

Examples

```
print(just_anything2(inpt_v = c("oui222jj44", "oui122jj"),
    symbol_ = "-", anything_v = letters))
[1] "oui---jj--" "oui---jj"
```

just_anything3

just_anything3

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_anything3(inpt_v, anything_v = c())
```

Arguments

inpt_v

is the input vector

Examples

```
print(just_anything3(inpt_v = c("oui222jj644", "oui122jj"),
    anything_v = letters))
[1] "ouijj" "ouijj"
```

just_chr

just_chr

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_chr(inpt_v, symbol_ = "-")
```

Arguments

inpt_v is the input vector

symbol_ is the chosen symbol to replace numbers

just_chr2 101

Examples

just_chr2

just_chr2

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_chr2(inpt_v, symbol_ = "-")
```

Arguments

inpt_v is the input vector
symbol_ is the chosen symbol to replace numbers

Examples

just_chr3

just_chr3

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_chr3(inpt_v)
```

Arguments

inpt_v

is the input vector

```
print(just_chr3(inpt_v = c("oui222jj644", "oui122jj")))
[1] "ouijj" "ouijj"
```

102 just_nb2

just_nb just_nb

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_nb(inpt_v, symbol_ = "-")
```

Arguments

```
inpt_v is the input vector
symbol_ is the chosen symbol to replace numbers
```

Examples

just_nb2

just_nb2

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_nb2(inpt_v, symbol_ = "-")
```

Arguments

```
inpt_v is the input vector
symbol_ is the chosen symbol to replace numbers
```

```
print(just_nb2(inpt_v = c("oui222jj44", "oui122jj"),
        symbol_ = "-"))

[1] "---222--44" "---122--"
```

just_nb3 103

just_nb3

Description

Extract only the letters from all elements of a vector, see examples

just_nb3

Usage

```
just_nb3(inpt_v)
```

Arguments

inpt_v is the input vector

Examples

```
print(just_nb3(inpt_v = c("oui222jj644", "oui122jj")))
[1] 222644 122
```

```
just_not_anything just_not_anything
```

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_not_anything(inpt_v, symbol_ = "-", anything_v = c())
```

Arguments

```
inpt_v is the input vector
symbol_ is the chosen symbol to replace numbers
```

just_not_anything3

```
just_not_anything2 just_not_anything2
```

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_not_anything2(inpt_v, symbol_ = "-", anything_v = c())
```

Arguments

```
inpt_v is the input vector
symbol_ is the chosen symbol to replace numbers
```

Examples

```
just_not_anything3 just_not_anything3
```

Description

Extract only the letters from all elements of a vector, see examples

Usage

```
just_not_anything3(inpt_v, anything_v = c())
```

Arguments

```
inpt_v is the input vector
```

leap_yr

leap_yr

leap_year

Description

Get if the year is leap

Usage

```
leap_yr(year)
```

Arguments

year

is the input year

Examples

```
print(leap_yr(year=2024))
#[1] TRUE
```

left_all

left_all

Description

Allow to apply left join on n dataframes, datatables, tibble

Usage

```
left_all(..., keep_val = FALSE, id_v)
```

Arguments

```
are all the dataframes etckeep_val is if you want to keep the id columnid_v is the common id of all the dataframes etc
```

```
datf1 <- data.frame(
    "id1"=c(1:5),
    "var1"=c("oui", "oui", "oui", "non", "non")
)

datf2 <- data.frame(
    "id1"=c(1, 2, 3, 7, 9),
    "var1"=c("oui2", "oui2", "oui2", "non2", "non2")
)</pre>
```

106 list_files

```
print(left_all(datf1, datf2, datf2, datf2, keep_val=FALSE, id_v="id1"))
 id1 var1.x var1.y var1.x.x var1.y.y
      oui
           oui2 oui2
                         oui2
                  oui2
   2
           oui2
                           oui2
       oui
       oui oui2 oui2
   3
                          oui2
      non <NA> <NA>
4
                           <NA>
   5 non <NA> <NA> <NA># '
print(left_all(datf1, datf2, datf2, keep_val=FALSE, id_v="id1"))
 id1 var1.x var1.y var1
      oui oui2 oui2
  2
2
      oui oui2 oui2
3
  3 oui oui2 oui2
4
  4 non <NA> <NA>
  5 non <NA> <NA>
```

```
letter_to_nb
```

letter_to_nb

Description

Allow to get the number of a spreadsheet based column by the letter ex: AAA = 703

Usage

```
letter_to_nb(letter)
```

Arguments

```
letter is the letter (name of the column)
```

Examples

```
print(letter_to_nb("rty"))
#[1] 12713
```

```
list_files
```

list_files

Description

A list.files() based function addressing the need of listing the files with extension a or or extension b ...

Usage

```
list_files(patternc, pathc = ".")
```

lst_flatnr 107

Arguments

patternc is a vector containing all the exensions you want
pathc is the path, can be a vector of multiple path because list.files() supports it.

Description

Flatten a list to a vector

Usage

```
lst_flatnr(inpt_l)
```

Arguments

inpt_l is the input list

Examples

```
print(lst_flatnr(inpt_l=list(c(1, 2), c(5, 3), c(7, 2, 7))))
#[1] 1 2 5 3 7 2 7
```

match_by

match_by

Description

Allow to match elements by ids, see examples.

Usage

```
match_by(to_match_v = c(), inpt_v = c(), inpt_ids = c())
```

Arguments

 $\verb"to_match_v" is the vector containing all the elements to match$

inpt_v is the input vector containing all the elements that could contains the elements

to match. Each elements is linked to an element from inpt_ids at any given

index, see examples. So inpt_v and inpt_ids must be the same size

inpt_ids is the vector containing all the ids for the elements in inpt_v. An element is

linked to the id x is both are at the same index. So inpt_v and inpt_ids must be

the same size

108 multitud

Examples

match_na_omit

match_na_omit

Description

Performs a match, but remove the NA values in the output if there is one or many, see examples.

Usage

```
match_na_omit(x, table)
```

Arguments

x is the vector of the patterns to be matched table is the vector that may contain the patterns to be matched

Examples

```
match_na_omit(x = c("oui", "non", "2"), table = c("1", "oui", "oui", "ee", "non"))
[1] 2 5
```

multitud

multitud

Description

```
From a list containing vectors allow to generate a vector following this rule: list(c("a", "b"), c("1", "2"), c("A", "Z", "E")) -> c("a1A", "b1A", "a2A", "b2A", a1Z, ...)
```

Usage

```
multitud(l, sep_ = "")
```

mutate_vector 109

Arguments

```
is the list

sep_ is the separator between elements (default is set to "" as you see in the example)
```

Examples

```
print(multitud(l=list(c("a", "b"), c("1", "2"), c("A", "Z", "E"), c("Q", "F")), sep_="/")
#[1] "a/1/A/Q" "b/1/A/Q" "a/2/A/Q" "b/2/A/Q" "a/1/Z/Q" "b/1/Z/Q" "a/2/Z/Q"
#[8] "b/2/Z/Q" "a/1/E/Q" "b/1/E/Q" "a/2/E/Q" "b/2/E/Q" "a/1/A/F" "b/1/A/F"
#[15] "a/2/A/F" "b/2/A/F" "a/1/Z/F" "b/1/Z/F" "a/2/Z/F" "b/2/Z/F" "a/1/E/F"
#[22] "b/1/E/F" "a/2/E/F" "b/2/E/F"
```

mutate_vector

mutate_vector

Description

Allow to select elements from a vector according to a uniform distribution (better than runif, especially for small vectors). You can choose the seed that the reimplementation of the uniform distribution will work with, defaults to 'random_data2.csv', see examples

Usage

```
mutate_vector(
   inpt_v,
   n_inpt,
   nvr_here = "NULL",
   base_seed = "random_data2.csv"
)
```

Arguments

inpt_v is the input vector containing the elements to evenly select

n_inpt is the number of elements of the output vector
nvr_here is a known value that is never in 'inpt_v'

base_seed is a csv filename containing the values for the reimplentation of the uniform

function to work with

nb2_follow

```
length(x)

[1] 5000

datf <- occu(x)

datf %>%
   ggplot(mapping = aes(x = var, y = occurence)) +
   geom_col() +
   theme_minimal()
```

nb2 follow

nb2_follow

Description

Allows to get the number and pattern of potential continuous pattern after an index of a vector, see examples

Usage

```
nb2_follow(inpt_v, inpt_idx, inpt_follow_v = c())
```

Arguments

```
inpt_v is the input vector
inpt_idx is the index
inpt_follow_v
```

is a vector containing the patterns that are potentially just after inpt_nb

```
print(nb2_follow(inpt_v = c(1:12), inpt_idx = 4, inpt_follow_v = c(5)))

[1] 1 5
# we have 1 times the pattern 5 just after the 4nth index of inpt_v

print(nb2_follow(inpt_v = c(1, "non", "oui", "oui", "oui", "nop", 5), inpt_idx = 2, inpt_
[1] "3" "oui"

# we have 3 times continuously the pattern 'oui' and 0 times the pattern 5 just after the print(nb2_follow(inpt_v = c(1, "non", "5", "5", "nop", 5), inpt_idx = 2, inpt_follow
[1] "3" "5"
```

nb_follow 111

nb_follow

nb_follow

Description

Allow to get the number of certains patterns that may be after an index of a vector continuously, see examples

Usage

```
nb_follow(inpt_v, inpt_idx, inpt_follow_v = c())
```

Arguments

```
inpt_v is the input vector
inpt_idx is the index
inpt_follow_v
```

is a vector containing all the potential patterns that may follow the element in the vector at the index inpt_idx

Examples

nb_to_letter

 nb_to_letter

Description

Allow to get the letter of a spreadsheet based column by the number ex: 703 = AAA

Usage

```
nb_to_letter(x)
```

Arguments

Х

is the number of the column

nb_to_letter

```
print(nb_to_letter(5))
[1] "e"
print(nb_to_letter(27))
[1] "aa"
print(nb_to_letter(51))
[1] "ay"
print(nb_to_letter(52))
[1] "az"
print(nb_to_letter(53))
[1] "ba"
print(nb_to_letter(675))
[1] "yy"
print(nb_to_letter(676))
[1] "yz"
print(nb_to_letter(677))
[1] "za"
print(nb_to_letter(702))
[1] "zz"
print(nb_to_letter(703))
[1] "aaa"
print(nb_to_letter(18211))
[1] "zxk"
print(nb_to_letter(18277))
[1] "zzy"
print(nb_to_letter(18278))
[1] "zzz"
print(nb_to_letter(18279))
[1] "aaaa"
```

nestr_datf1

nestr_datf1

nestr_datf1

Description

Allow to write a value (1a) to a dataframe (1b) to its cells that have the same coordinates (row and column) than the cells whose value is equal to a another special value (2a), from another another dataframe (2b). The value (1a) depends of the cell value coordinates of the third dataframe (3b). If a cell coordinates (1c) of the first dataframe (1b) does not correspond to the coordinates of a good returning cell value (2a) from the dataframe (2b), so this cell (1c) can have its value changed to the same cell coordinates value (3a) of a third dataframe (4b), if (4b) is not set to NA.

Usage

```
nestr_datf1(
   inptf_datf,
   inptt_pos_datf,
   nestr_datf,
   yes_val = TRUE,
   inptt_neg_datf = NA
)
```

Arguments

```
print(nestr_datf1(inptf_datf=data.frame(c(1, 2, 1), c(1, 5, 7)),
inptt_pos_datf=data.frame(c(4, 4, 3), c(2, 1, 2)),
inptt_neg_datf=data.frame(c(44, 44, 33), c(12, 12, 12)),
nestr_datf=data.frame(c(TRUE, FALSE, TRUE), c(FALSE, FALSE, TRUE)), yes_val=TRUE))
  c.1..2..1. c.1..5..7.
#1
           4
                      12
                      12
#2
           44
            3
#3
print(nestr_datf1(inptf_datf=data.frame(c(1, 2, 1), c(1, 5, 7)),
inptt_pos_datf=data.frame(c(4, 4, 3), c(2, 1, 2)),
inptt_neg_datf=NA,
nestr_datf=data.frame(c(TRUE, FALSE, TRUE), c(FALSE, FALSE, TRUE)), yes_val=TRUE))
    c.1..2..1. c.1..5..7.
```

114 nest_v

```
#1
              4
#2
              2
#3
```

|--|--|

Description

Allow to write a special value (1a) in the cells of a dataframe (1b) that correspond (row and column) to whose of another dataframe (2b) that return another special value (2a). The cells whose coordinates do not match the coordinates of the dataframe (2b), another special value can be written (3a) if not set to NA.

Usage

```
nestr_datf2(inptf_datf, rtn_pos, rtn_neg = NA, nestr_datf, yes_val = T)
```

Arguments

```
is the input dataframe (1b)
inptf_datf
                  is the special value (1a)
rtn_pos
                  is the special value (3a)
rtn_neg
                 is the dataframe (2b)
nestr_datf
                  is the special value (2a)
yes_val
```

Examples

```
 \texttt{print} (\texttt{nestr\_datf2} (\texttt{inptf\_datf=data.frame} (\texttt{c(1, 2, 1)}, \ \texttt{c(1, 5, 7)}), \ \texttt{rtn\_pos="yes", results of the content of the cont
rtn_neg="no", nestr_datf=data.frame(c(TRUE, FALSE, TRUE), c(FALSE, FALSE, TRUE)), yes_val
  # c.1..2..1. c.1..5..7.
  #1
                                                                                         yes
                                                                                                                                                                                                                         no
  #2
                                                                                                         no
                                                                                                                                                                                                                           no
  #3
                                                                                                   yes
```

```
nest\_v
nest_v
```

Description

Nest two vectors according to the following parameters.

yes

Usage

```
nest_v(f_v, t_v, step = 1, after = 1)
```

new_ordered 115

Arguments

f_v	is the vector that will welcome the nested vector t_v
t_v	is the imbriquator vector
step	defines after how many elements of f_v the next element of t_v can be put in the output
after	defines after how many elements of f v, the beginning of t v can be put

Examples

```
print(nest_v(f_v=c(1, 2, 3, 4, 5, 6), t_v=c("oui", "oui2", "oui3", "oui4", "oui5", "oui6"
    step=2, after=2))
#[1] "1" "2" "oui" "3" "4" "oui2" "5" "6" "oui3" "oui4"
```

Description

Returns the indexes of elements contained in "w_v" according to "f_v"

Usage

```
new_ordered(f_v, w_v, nvr_here = NA)
```

Arguments

f_v is the input vector
w_v is the vector containing the elements that can be in f_v
nvr_here is a value you are sure is not present in f_v

```
print(new_ordered(f_v=c("non", "non", "oui"), w_v=c("oui", "non", "non")))
#[1] 4 1 2
```

normal_offset_prob

normal_dens	normal_dens
-------------	-------------

Description

Calculates the normal distribution probality, see examples

Usage

```
normal_dens(target_v = c(), mean, sd, step = 0.01)
```

Arguments

target_v is the input interval, see examples
mean is the mean of the normal distribution

is the standard deviation of the normal distribution

step is the step with which the result will be outputed, the smaller the step is, the

more accurate the result will be

Examples

```
print (normal_dens(target_v = c(5, 19), mean = 12, sd = 1.5, step = 0.1))
[1] 0.9999974
print (normal_dens(target_v = c(10.5, 13.5), mean = 12, sd = 1.5, step = 0.1))
[1] 0.6986416
print (normal_dens(target_v = c(10.5, 13.5), mean = 12, sd = 1.5, step = 0.01))
[1] 0.6843008
```

```
normal_offset_prob normal_offset_prob
```

Description

Returns the probability of the most offset value in the input vector accrding to the normal distribution, see examples.

Usage

```
normal_offset_prob(inpt_v = c(), mean_inpt, sd_inpt)
```

Arguments

mean_inpt is the mean of the normal distribution
sd_inpt is the standard deviations of the normal distribution

normal_offset_val 117

Examples

```
normal_offset_val normal_offset_val
```

Description

Returns the most offset value from an normaldistribution, given tha probabilities, see examples.

Usage

```
normal_offset_val(mean_inpt, sd_inpt, proba = 0.01)
```

Arguments

mean_inpt is the mean of the normal distribution

sd_inpt is the standard deviation of the normal distribution

proba is the probabilitie of the most offset value to include in the normal distribution

```
normal_offset_val(mean_inpt = 12, sd_inpt = 3, proba = 0.01)
[1] 18.82475
## offset values are 12 + (18.82475 - 12) and 12 - (18.82475 - 12)
normal_offset_val(mean_inpt = 18, sd_inpt = 1.2, proba = 0.01)
[1] 21.17667
## ## offset values are 18 + (21.17667 - 18) and 18 - (21.17667 - 18)
```

118 old_to_new_idx

occu occu

Description

Allow to see the occurence of each variable in a vector. Returns a datafame with, as the first column, the all the unique variable of the vector and, in he second column, their occurence respectively.

Usage

```
occu(inpt_v)
```

Arguments

```
inpt_v the input dataframe
```

Examples

Description

Allow to convert index of elements in a vector $inpt_v$ to index of an vector type 1:sum(nchar(inpt_v)), see examples

Usage

```
old_to_new_idx(inpt_v = c())
```

Arguments

```
inpt_v is the input vector
```

```
print(old_to_new_idx(inpt_v = c("oui", "no", "eeee")))
[1] 1 1 1 2 2 3 3 3 3
```

or_bool1 119

or_bool1 or_bool1

Description

Returns a boolean vector according to 'or' condition on boolean vector in a dataframe, each column represents a boolean vector, see examples, uses a different algorithm than or_bool1

Usage

```
or_bool1(inpt_datf)
```

Examples

or_bool2

 or_bool2

Description

Returns a boolean vector according to 'or' condition on boolean vector in a dataframe, each column represents a boolean vector, see examples, uses a different algorithm than or_bool1

Usage

```
or_bool2(inpt_datf)
```

120 pairs_findr_merger

pairs	findr	pairs_	findr

Description

Takes a character as input and detect the pairs of pattern, like the parenthesis pais if the pattern is "(" and then ")"

Usage

```
pairs_findr(inpt, ptrn1 = "(", ptrn2 = ")")
```

Arguments

```
inpt is the input character
ptrn1 is the first pattern ecountered in the pair
ptrn2 is the second pattern in the pair
```

Examples

```
print (pairs_findr(inpt="ze+(yu*45/(jk+zz)*(o()p))-(re*(rt+qs)-fg)"))
[[1]]
[1] 4 1 1 3 2 2 3 4 6 5 5 6
[[2]]
[1] 4 11 17 19 21 22 24 25 27 31 37 41
```

```
pairs_findr_merger pairs_findr_merger
```

Description

Takes two different outputs from pairs_findr and merge them. Can be usefull when the pairs consists in different patterns, for example one output from the pairs_findr function with ptrn1 = "(" and ptrn2 = ")", and a second output from the pairs_findr function with ptrn1 = "" and ptrn2 = "".

Usage

```
pairs_findr_merger(lst1 = list(), lst2 = list())
```

Arguments

lst1	is the first ouput from pairs findr function
lst2	is the second ouput from pairs findr function

pairs_findr_merger 121

```
print(pairs_findr_merger(lst1=list(c(1, 2, 3, 3, 2, 1), c(3, 4, 5, 7, 8, 9)),
                        lst2=list(c(1, 1), c(1, 2)))
[[1]]
[1] 1 1 2 3 4 4 3 2
[[2]]
[1] 1 2 3 4 5 7 8 9
print(pairs_findr_merger(lst1=list(c(1, 2, 3, 3, 2, 1), c(3, 4, 5, 7, 8, 9))),
                         lst2=list(c(1, 1), c(1, 11)))
[[1]]
[1] 1 2 3 4 4 3 2 1
[[2]]
[1] 1 3 4 5 7 8 9 11
print (pairs\_findr\_merger(lst1=list(c(1, 2, 3, 3, 2, 1), c(3, 4, 5, 8, 10, 11)),\\
                        lst2=list(c(4, 4), c(6, 7)))
[[1]]
[1] 1 2 3 4 4 3 2 1
[[2]]
[1] 3 4 5 6 7 8 10 11
print(pairs_findr_merger(lst1=list(c(1, 2, 3, 3, 2, 1), c(3, 4, 5, 7, 10, 11)),
                         lst2=list(c(4, 4), c(8, 9))))
[[1]]
[1] 1 2 3 3 4 4 2 1
[[2]]
[1] 3 4 5 7 8 9 10 11
print(pairs_findr_merger(lst1=list(c(1, 2, 3, 3, 2, 1), c(3, 4, 5, 7, 10, 11))),\\
                         lst2=list(c(4, 4), c(18, 19)))
[[1]]
[1] 1 2 3 3 2 1 4 4
[[2]]
[1] 3 4 5 7 10 11 18 19
print(pairs_findr_merger(lst1 = list(c(1, 1, 2, 2, 3, 3), c(1, 25, 26, 32, 33, 38)),
                       lst2 = list(c(1, 1, 2, 2, 3, 3), c(7, 11, 13, 17, 19, 24))))
[[1]]
[1] 1 2 2 3 3 4 4 1 5 5 6 6
[1] 1 7 11 13 17 19 24 25 26 32 33 38
print(pairs_findr_merger(lst1 = list(c(1, 1, 2, 2, 3, 3), c(2, 7, 9, 10, 11, 15)),
```

122 pairs_insertr

pairs_insertr

pairs_insertr

Description

Takes a character representing an arbitrary condition (like ReGeX for example) or an information (to a parser for example), vectors containing all the pair of pattern that potentially surrounds condition (flagged_pair_v and corr_v), and a vector containing all the conjuntion character, as input and returns the character with all or some of the condition surrounded by the pair characters. See examples. All the pair characters are inserted according to the closest pair they found priotizing those found next to the condition and on the same depth-level and , if not found, the pair found at the n+1 depth-level.

Usage

```
pairs_insertr(
  inpt,
  algo_used = c(1:3),
  flagged_pair_v = c(")", "]"),
  corr_v = c("(", "["),
  flagged_conj_v = c("&", "|")
)
```

Arguments

inpt

is the input character representing an arbitrary condition, like ReGex for example, or information to a parser for example

algo_used

is a vector containing one or more of the 3 algorythms used. The first algorythm will simply put the pair of parenthesis at the condition surrounded and/or after a character flagged (in flagged_conj_v) as a conjunction. The second algorythm will put parenthesis at the condition that are located after other conditions that are surrounded by a pair. The third algorythm will put a pair at all the condition, it is very powerfull but takes a longer time. See examples and make experience to see which combination of algorythm(s) is the most efficient for your use case.

pairs_insertr2 123

```
flagged_pair_v
is a vector containing all the first character of the pairs
corr_v
is a vector containing all the last character of the pairs
flagged_conj_v
is a vector containing all the conjunction character
```

Examples

```
print(pairs_insertr(inpt = "([one]|two|twob)three(four)", algo_used = c(1)))
[1] "([one]|[two]|[twob])three(four)"
print(pairs_insertr(inpt = "(one|[two]|twob)three(four)", algo_used = c(2)))
[1] "(one|[two]|[twob])(three)(four)"
print(pairs_insertr(inpt = "(oneA|[one]|two|twob)three(four)", algo_used = c(1, 2)))
[1] "(oneA|[one]|[two]|[twob])(three)(four)"
print(pairs_insertr(inpt = "(oneA|[one]|two|twob)three(four)", algo_used = c(1, 2, 3)))
[1] "([oneA]|[one]|[two]|[twob])(three)(four)"
print(pairs_insertr(inpt = "(oneA|[one]|two|twob)three(four)", algo_used = c(3)))
[1] "([oneA]|[one]|(two)|(twob))(three)(four)"
print(pairs_insertr(inpt = "(oneA|[one]|two|twob)three((four))", algo_used = c(3)))
[1] "([oneA]|[(one)]|(two)|(twob))(three)((four))"
```

Description

Takes a character representing an arbitrary condition (like ReGeX for example) or an information (to a parser for example), vectors containing all the pair of pattern that potentially surrounds condition (flagged_pair_v and corr_v), and a vector containing all the conjuntion character, as input and returns the character with all or some of the condition surrounded by the pair characters. See examples. All the pair characters are inserted according to the closest pair they found priotizing those found next to the condition and on the same depth-level and , if not found, the pair found at the n+1 depth-level.

Usage

```
pairs_insertr2(
  inpt,
  algo_used = c(1:3),
  flagged_pair_v = c(")", "]"),
  corr_v = c("(", "["),
```

124 pairs_insertr2

```
flagged_conj_v = c("&", "|"),
method = c("(", ")")
)
```

Arguments

inpt is the input character representing an arbitrary condition, like ReGex for exam-

ple, or information to a parser for example

algo_used is a vector containing one or more of the 3 algorythms used. The first algorythm

will simply put the pair of parenthesis at the condition surrounded and/or after a character flagged (in flagged_conj_v) as a conjunction. The second algorythm will put parenthesis at the condition that are located after other conditions that are surrounded by a pair. The third algorythm will put a pair at all the condition, it is very powerfull but takes a longer time. See examples and make experience to see which combination of algorythm(s) is the most efficient for your use case.

flagged_pair_v

is a vector containing all the first character of the pairs

corr_v is a vector containing all the last character of the pairs

flagged_conj_v

is a vector containing all the conjunction character

method is length 2 vector containing as a first index, the first character of the pair in-

serted, and at the last index, the second and last character of the pair

```
print(pairs_insertr2(inpt = "([one]|two|twob)three(four)", algo_used = c(1), method = c('[1] "([one]|(two)|(twob))three(four)"
print(pairs_insertr2(inpt = "([one]|two|twob)three(four)", algo_used = c(1), method = c('[1] "([one]|[two]|[twob])three(four)"
print(pairs_insertr2(inpt = "(oneA|[one]|two|twob)three(four)", algo_used = c(1, 2)))
[1] "(oneA|[one]|(two)|(twob)) (three) (four)"
print(pairs_insertr2(inpt = "(oneA|[one]|two|twob)three(four)", algo_used = c(1, 2), method flagged_pair_v = c(")", "]", "#"), corr_v = c("(", "[", "-")))
[1] "(oneA|[one]|-two#|-twob#)-three#(four)"
print(pairs_insertr2(inpt = "(oneA|[one]|two|twob)three(four)", algo_used = c(1, 2, 3)))
[1] "((oneA)|[one]|(two)|(twob)) (three) (four)"
print(pairs_insertr2(inpt = "(oneA|[one]|two|twob)three(four)", algo_used = c(3), method [1] "([oneA]|[one]|[two]|[twob]) [three] (four)"
print(pairs_insertr2(inpt = "(oneA|[one]|two|twob)three((four))", algo_used = c(3)))
[1] "((oneA)|[one]|(two)|(twob)) (three) ((four))"
```

paste_datf 125

```
paste_datf
```

Description

Return a vector composed of pasted elements from the input dataframe at the same index.

Usage

```
paste_datf(inpt_datf, sep = "")
```

Arguments

```
inpt_datf is the input dataframe
sep is the separator between pasted elements, defaults to ""
```

paste_datf

Examples

```
print(paste_datf(inpt_datf=data.frame(c(1, 2, 1), c(33, 22, 55))))
[1] "133" "222" "155"
```

Description

Return a vector composed of pasted elements from the input dataframe at the same column.

Usage

```
paste_datf2(inpt_datf, sep = "")
```

Arguments

```
inpt_datf is the input dataframe
sep is the separator between pasted elements, defaults to ""
```

```
print(paste_datf2(inpt_datf=data.frame(c(1, 2, 1), c(33, 22, 55))))
#[1] "121" "332255"
```

126 pattern_gettr

```
pattern_generator pattern_generator
```

Description

Allow to create patterns which have a part that is varying randomly each time.

Usage

```
pattern_generator(base_, from_, nb, hmn = 1, after = 1, sep = "")
```

Arguments

base_	is the pattern that will be kept
from_	is the vector from which the elements of the random part will be generated
nb	is the number of random pattern chosen for the varying part
hmn	is how many of varying pattern from the same base will be created
after	is set to 1 by default, it means that the varying part will be after the fixed part, set to 0 if you want the varying part to be before
sep	is the separator between all patterns in the returned value

Examples

```
print(pattern_generator(base_="oui", from_=c("er", "re", "ere"), nb=1, hmn=3))
# [1] "ouier" "ouire" "ouier"
print(pattern_generator(base_="oui", from_=c("er", "re", "ere"), nb=2, hmn=3, after=0, set [1] "er-re-o-u-i" "ere-re-o-u-i" "ere-er-o-u-i"
```

Description

Search for pattern(s) contained in a vector in another vector and return a list containing matched one (first index) and their position (second index) according to these rules: First case: Search for patterns strictly, it means that the searched pattern(s) will be matched only if the patterns contained in the vector that is beeing explored by the function are present like this c("pattern_searched", "other", ..., "pattern_searched") and not as c("other_thing pattern_searched other_thing", "other", ..., "pattern_searched other_thing") Second case: It is the opposite to the first case, it means that if the pattern is partially present like in the first position and the last, it will be considered like a matched pattern. REGEX can also be used as pattern

pattern_tuning 127

Usage

```
pattern_gettr(
  word_,
  vct,
  occ = c(1),
  strict,
  btwn,
  all_in_word = "yes",
  notatall = "###"
)
```

Arguments

word_	is the vector containing the patterns
vct	is the vector being searched for patterns
occ	a vector containing the occurence of the pattern in word_ to be matched in the vector being searched, if the occurence is 2 for the nth pattern in word_ and only one occurence is found in vct so no pattern will be matched, put "forever" to no longer depend on the occurence for the associated pattern
strict	a vector containing the "strict" condition for each nth vector in word_ ("strict" is the string to activate this option)
btwn	is a vector containing the condition ("yes" to activate this option) meaning that if "yes", all elements between two matched patern in vct will be returned , so the patterns you enter in word_ have to be in the order you think it will appear in vct
all_in_word	is a value (default set to "yes", "no" to activate this option) that, if activated, won't authorized a previous matched pattern to be matched again
notatall	is a string that you are sure is not present in vct

Examples

```
print(pattern_gettr(word_=c("oui", "non", "erer"), vct=c("oui", "oui", "non", "oui",
    "non", "opp", "opp", "erer", "non", "ok"), occ=c(1, 2, 1),
    btwn=c("no", "yes", "no"), strict=c("no", "no", "ee")))

#[[1]]
#[1] 1 5 8
#
#[[2]]
#[1] "oui" "non" "opp" "opp" "erer"
```

Description

Allow to tune a pattern very precisely and output a vector containing its variations n times.

128 power_to_char

Usage

```
pattern_tuning(
  pattrn,
  spe_nb,
  spe_l,
  exclude_type,
  hmn = 1,
  rg = c(1, nchar(pattrn))
)
```

Arguments

pattrn is the character that will be tuned

spe_nb is the number of new character that will be replaced

spe_1 is the source vector from which the new characters will replace old ones

exclude_type is character that won't be replaced

hmn is how many output the function will return

rg is a vector with two parameters (index of the first letter that will be replaced,

index of the last letter that will be replaced) default is set to all the letters from

the source pattern

Examples

```
print(pattern_tuning(pattrn="oui", spe_nb=2, spe_l=c("e", "r", "T", "O"), exclude_type="of"
#[1] "orT" "oTr" "oOi"
```

power_to_char

power_to_char

Description

Convert a scientific number to a string representing normally the number.

Usage

```
power_to_char(inpt_v = c())
```

Arguments

inpt_v is the input vector containing scientific number, but also other elements that

won't be taken in count

```
print(power_to_char(inpt_v = c(22 * 10000000, 12, 9 * 0.0000002)))
[1] "2200000000" "12" "0.0000018"
```

pre_to_post_idx 129

Description

Allow to convert indexes from a pre-vector to post-indexes based on a current vector, see examples

Usage

```
pre_to_post_idx(inpt_v = c(), inpt_idx = c(1:length(inppt_v)))
```

Arguments

```
inpt_v is the new vector
inpt_idx is the vector containing the pre-indexes
```

Examples

```
print(pre_to_post_idx(inpt_v = c("oui", "no", "eee"), inpt_idx = c(1:8)))
[1] 1 1 1 2 2 3 3 3
As if the first vector was c("o", "u", "i", "n", "o", "e", "e", "e")
```

```
ptrn_switchr ptrn_switchr
```

Description

Allow to switch, copy pattern for each element in a vector. Here a pattern is the values that are separated by a same separator. Example: "xx-xxx-xx" or "xx/xx/xxxx". The xx like values can be switched or copied from whatever index to whatever index. Here, the index is like this 1-2-3 etcetera, it is relative of the separator.

Usage

```
ptrn_switchr(inpt_l, f_idx_l = c(), t_idx_l = c(), sep = "-", default_val = NA)
```

Arguments

inpt_l	is the input vector
f_idx_l	is a vector containing the indexes of the pattern you want to be altered.
t_idx_l	is a vector containing the indexes to which the indexes in f_idx_l are related.
sep	is the separator, defaults to "-"
default_val	is the default value, if not set to NA, of the pattern at the indexes in f_idx_l. If it is not set to NA, you do not need to fill t_idx_l because this is the vector containing the indexes of the patterns that will be set as new values relatively to the indexes in f_idx_l. Defaults to NA.

ptrn_twkr

Examples

```
print (ptrn_switchr(inpt_l=c("2022-01-11", "2022-01-14", "2022-01-21",
   "2022-01-01"), f_idx_l=c(1, 2, 3), t_idx_l=c(3, 2, 1)))
#[1] "11-01-2022" "14-01-2022" "21-01-2022" "01-01-2022"
print (ptrn_switchr(inpt_l=c("2022-01-11", "2022-01-14", "2022-01-21",
   "2022-01-01"), f_idx_l=c(1), default_val="ee"))
#[1] "ee-01-11" "ee-01-14" "ee-01-21" "ee-01-01"
```

ptrn_twkr

ptrn_twkr

Description

Allow to modify the pattern length of element in a vector according to arguments. What is here defined as a pattern is something like this xx-xx-xx or xx/xx/xxx... So it is defined by the separator

Usage

```
ptrn_twkr(
   inpt_l,
   depth = "max",
   sep = "-",
   default_val = "0",
   add_sep = TRUE,
   end_ = TRUE
)
```

Arguments

inpt_l	is the input vector
depth	is the number (numeric) of separator it will keep as a result. To keep the number of separator of the element that has the minimum amount of separator do depth="min" and depth="max" (character) for the opposite. This value defaults to "max".
sep	is the separator of the pattern, defaults to "-"
default_val	is the default val that will be placed between the separator, defaults to "00"
add_sep	defaults to TRUE. If set to FALSE, it will remove the separator for the patterns that are included in the interval between the depth amount of separator and the actual number of separator of the element.
end_	is if the default_val will be added at the end or at the beginning of each element that lacks length compared to depth

read_edm_parser 131

Examples

```
v <- c("2012-06-22", "2012-06-23", "2022-09-12", "2022")

ptrn_twkr(inpt_l=v, depth="max", sep="-", default_val="00", add_sep=TRUE)

#[1] "2012-06-22" "2012-06-23" "2022-09-12" "2022-00-00"

ptrn_twkr(inpt_l=v, depth=1, sep="-", default_val="00", add_sep=TRUE)

#[1] "2012-06" "2012-06" "2022-09" "2022-00"

ptrn_twkr(inpt_l=v, depth="max", sep="-", default_val="00", add_sep=TRUE, end_=FALSE)

#[1] "2012-06-22" "2012-06-23" "2022-09-12" "00-00-2022"</pre>
```

read_edm_parser

Description

Allow to read data from edm parsed dataset, see examples

Usage

```
read_edm_parser(inpt, to_find_v = c())
```

Arguments

inpt is the input dataset
to_find_v is the vector containing the path to find the data, see examples

```
print (read_edm_parser("(ok(ee:56)) (ok(oui(rr((rr2:6)(rr:5)))) (oui(bb(rr2:1))) (ee1:4))",
to_find_v = c("ok", "oui", "rr", "rr2")))

[1] "6"

print (read_edm_parser("(ok(ee:56)) (ok(oui(rr((rr2:6)(rr:5)))) (oui(bb(rr2:1))) (ee1:4))", t

[1] "56"

print (read_edm_parser("(ok(ee:56)) (ok(oui(rr((rr2:6)(rr:5)))) (oui(bb(rr2:1))) (ee1:4))", t

[1] "56"
```

132 regex_spe_detect

rearangr_v	rearangr_v

Description

Reanranges a vector "w_v" according to another vector "inpt_v". inpt_v contains a sequence of number. inpt_v and w_v have the same size and their indexes are related. The output will be a vector containing all the elements of w_v rearanges in descending or asending order according to inpt_v

Usage

```
rearangr_v(inpt_v, w_v, how = "increasing")
```

Arguments

inpt_v is the vector that contains the sequence of number
 w_v is the vector containing the elements related to inpt_v
 how is the way the elements of w_v will be outputed according to if inpt_v will be sorted ascendigly or descendingly

Examples

```
print(rearangr_v(inpt_v=c(23, 21, 56), w_v=c("oui", "peut", "non"), how="decreasing"))
#[1] "non" "oui" "peut"
```

Description

Takes a character as input and returns its regex-friendly character for R.

Usage

```
regex_spe_detect(inpt)
```

Arguments

inpt the input character

regroupr 133

Examples

```
print(regex_spe_detect("o"))
[1] "o"
print(regex_spe_detect("("))
[1] "\\(")
print(regex_spe_detect("tr(o)m"))
[1] "tr\\(o\\)m"
print(regex_spe_detect(inpt="fggfg[fggf]fgfg(vg?fgfgf.gf)"))
[1] "fggfg\\[fggf\\]fgfg\\(vg\\?fgfgf\\.gf\\)"
print(regex_spe_detect(inpt = "---"))
[1] "\\-\\-\\-"
```

regroupr

regroupr

Description

Allow to sort data like "c(X1/Y1/Z1, X2/Y1/Z2, ...)" to what you want. For example it can be to "c(X1/Y1/21, X1/Y1/Z2, ...)"

Usage

```
regroupr(
  inpt_v,
  sep_ = "-",
  order = c(1:length(unlist(strsplit(x = inpt_v[1], split = sep_)))),
  l_order = NA
)
```

Arguments

inpt_v	is the input vector containing all the data you want to sort in a specific way. All the sub-elements should be separated by a unique separator such as "-" or "/"
sep_	is the unique separator separating the sub-elements in each elements of inpt_v
order	is a vector describing the way the elements should be sorted. For example if you want this dataset " $c(X1/Y1/Z1, X2/Y1/Z2,)$ " to be sorted by the last element you should have order= $c(3:1)$, for example, and it should returns something like this $c(X1/Y1/Z1, X2/Y1/Z1, X1/Y2/Z1,)$ assuming you have only two values for X.
l_order	is a list containing the vectors of values you want to order first for each sub- elements

rm_na_rows

Examples

```
vec <- multitud(l=list(c("a", "b"), c("1", "2"), c("A", "Z", "E"), c("Q", "F")), sep_="/"</pre>
print (vec)
# [1] "a/1/A/Q" "b/1/A/Q" "a/2/A/Q" "b/2/A/Q" "a/1/Z/Q" "b/1/Z/Q" "a/2/Z/Q"
# [8] "b/2/Z/Q" "a/1/E/Q" "b/1/E/Q" "a/2/E/Q" "b/2/E/Q" "a/1/A/F" "b/1/A/F"
#[15] "a/2/A/F" "b/2/A/F" "a/1/Z/F" "b/1/Z/F" "a/2/Z/F" "b/2/Z/F" "a/1/E/F"
#[22] "b/1/E/F" "a/2/E/F" "b/2/E/F"
print(regroupr(inpt_v=vec, sep_="/"))
# [1] "a/1/1/1"
                "a/1/2/2"
                            "a/1/3/3"
                                       "a/1/4/4"
                                                  "a/1/5/5"
                                                             "a/1/6/6"
                          # [7] "a/2/7/7"
                "a/2/8/8"
#[13] "b/1/13/13" "b/1/14/14" "b/1/15/15" "b/1/16/16" "b/1/17/17" "b/1/18/18"
#[19] "b/2/19/19" "b/2/20/20" "b/2/21/21" "b/2/22/22" "b/2/23/23" "b/2/24/24"
vec <- vec[-2]
print(regroupr(inpt_v=vec, sep_="/"))
                                       "a/1/4/4"
# [1] "a/1/1/1"
                "a/1/2/2"
                           "a/1/3/3"
                                                  "a/1/5/5"
                                                             "a/1/6/6"
# [7] "a/2/7/7"
               "a/2/8/8"
                          #[13] "b/1/13/13" "b/1/14/14" "b/1/15/15" "b/1/16/16" "b/1/17/17" "b/2/18/18"
#[19] "b/2/19/19" "b/2/20/20" "b/2/21/21" "b/2/22/22" "b/2/23/23"
print(regroupr(inpt_v=vec, sep_="/", order=c(4:1)))
               "2/2/A/Q"
#[1] "1/1/A/Q"
                           "3/3/A/Q"
                                      "4/4/A/Q" "5/5/Z/Q" "6/6/Z/Q"
                          "9/9/E/Q" "10/10/E/Q" "11/11/E/Q" "12/12/E/Q"
# [7] "7/7/Z/Q"
               "8/8/Z/Q"
#[13] "13/13/A/F" "14/14/A/F" "15/15/A/F" "16/16/A/F" "17/17/Z/F" "18/18/Z/F"
#[19] "19/19/Z/F" "20/20/Z/F" "21/21/E/F" "22/22/E/F" "23/23/E/F" "24/24/E/F"
```

```
rm_na_rows rm_na_rows
```

Description

Allow to remove certain rows that contains NA, see examples.

Usage

```
rm_na_rows(inpt_datf, flagged_vals = c())
```

Arguments

```
inpt_datf is the input dataframe
```

flagged_vals is a vector containing the characters that will drop any rows that contains it

rm_rows 135

Examples

```
datf \leftarrow data.frame(c(1, 2, NA, 4), c(1:4))
print(datf)
  c.1..2..NA..4. c.1.4.
1
               1
2
               2
3
                       3
               NA
                4
                       4
print(rm_na_rows(inpt_datf = datf))
  c.1..2..NA..4. c.1.4.
1
               1
2
                2
                       2
4
                4
```

rm_rows

rm_rows

Description

Allow to remove certain rows that contains certains characters, see examples.

Usage

```
rm_rows(inpt_datf, flagged_vals = c())
```

Arguments

inpt_datf is the input dataframe

flagged_vals is a vector containing the characters that will drop any rows that contains it

```
datf \leftarrow data.frame(c(1, 2, NA, 4), c(1:4))
print(datf)
  c.1..2..NA..4. c.1.4.
1
               1
2
               2
3
                      3
              NA
4
               4
print(rm_rows(inpt_datf = datf, flagged_vals = c(1, 4)))
  c.1..2..NA..4. c.1.4.
2
               2
3
                     3
              NA
```

r_print

row_to_col

row_to_col

Description

Allow to reverse a dataframe (rows become cols and cols become rows)

Usage

```
row_to_col(inpt_datf)
```

Arguments

```
inpt_datf is the inout dataframe
```

Examples

```
datf_{test} \leftarrow data.frame(c(1, 11), c(2, 10), c(3, 9), c(4, 8))
print(datf_test)
 c.1..11. c.2..10. c.3..9. c.4..8.
1
       1
                2
                      3
       11
                10
                        9
                                 8
2
print(row_to_col(inpt_datf = datf_test))
 1 2
1 1 11
2 2 10
3 3 9
4 4
     8
```

 r_print

r_print

Description

Allow to print vector elements in one row.

Usage

```
r_print(inpt_v, sep_ = "and", begn = "This is", end = ", voila!")
```

Arguments

```
inpt_v is the input vector
```

sep_ is the separator between each elements

begin is the character put at the beginning of the print

end is the character put at the end of the print

save_untl 137

Examples

```
print(r_print(inpt_v=c(1:33)))  
#[1] "This is 1 and 2 and 3 and 4 and 5 and 6 and 7 and 8 and 9 and 10 and 11 and 12 and  
#and 14 and 15 and 16 and 17 and 18 and 19 and 20 and 21 and 22 and 23 and 24 and 25 and  
#and 27 and 28 and 29 and 30 and 31 and 32 and 33 and , voila!"
```

save_untl

save_untl

Description

Get the elements in each vector from a list that are located before certain values

Usage

```
save_untl(inpt_l = list(), val_to_stop_v = c())
```

Arguments

```
\begin{array}{ll} \text{inpt\_l} & \text{is the input list containing all the vectors} \\ \text{val\_to\_stop\_v} \end{array}
```

is a vector containing the values that marks the end of the vectors returned in the returned list, see the examples

```
print(save_untl(inpt_l=list(c(1:4), c(1, 1, 3, 4), c(1, 2, 4, 3)), val_to_stop_v=c(3, 4))
#[[1]]
#[1] 1 2
#
#[[2]]
#[1] 1 1
#
#[[3]]
#[1] 1 2
print(save_untl(inpt_l=list(c(1:4), c(1, 1, 3, 4), c(1, 2, 4, 3)), val_to_stop_v=c(3)))
#[[1]]
#[1] 1 2
#
#[[2]]
#[1] 1 2
#
#[[3]]
#[1] 1 2 4
```

138 see_datf

Description

Allow to return a dataframe with special value cells (ex: TRUE) where the condition entered are respected and another special value cell (ex: FALSE) where these are not

Usage

```
see_datf(
  datf,
  condition_l,
  val_l,
  conjunction_l = c(),
  rt_val = TRUE,
  f_val = FALSE
)
```

Arguments

dat.f is the input dataframe is the vector of the possible conditions ("==", ">", "<", "!=", "%%", "reg", condition 1 "not_reg", "sup_nchar", "inf_nchar", "nchar") (equal to some elements in a vector, greater than, lower than, not equal to, is divisible by, the regex condition returns TRUE, the regex condition returns FALSE, the length of the elements is strictly superior to X, the length of the element is strictly inferior to X, the length of the element is equal to one element in a vector), you can put the same condition n times. is the list of vectors containing the values or vector of values related to condival_l tion_l (so the vector of values has to be placed in the same order) conjunction_l contains the and or conjunctions, so if the length of condition 1 is equal to 3, there will be 2 conjunctions. If the length of conjunction_l is inferior to the length of condition_1 minus 1, conjunction_1 will match its goal length value with its last argument as the last arguments. For example, c("&", "I", "&") with a goal length value of $5 \rightarrow c("\&", "|", "\&", "\&", "\&")$ is a special value cell returned when the conditions are respected rt_val is a special value cell returned when the conditions are not respected f_val

Details

This function will return an error if number only comparative conditions are given in addition to having character values in the input dataframe.

```
datf1 <- data.frame(c(1, 2, 4), c("a", "a", "zu"))
print(see_datf(datf=datf1, condition_l=c("nchar"), val_l=list(c(1))))</pre>
```

see_diff

```
Х1
#1 TRUE TRUE
#2 TRUE TRUE
#3 TRUE FALSE
print(see\_datf(datf=datf1, condition\_l=c("=="), val\_l=list(c("a", 1)))))
    Х1
          Х2
#1 TRUE TRUE
#2 FALSE TRUE
#3 FALSE FALSE
print(see_datf(datf=datf1, condition_l=c("nchar"), val_l=list(c(1, 2))))
    X1
         X2
#1 TRUE TRUE
#2 TRUE TRUE
#3 TRUE TRUE
print(see_datf(datf=datf1, condition_l=c("not_reg"), val_l=list("[a-z]")))
    X1
          X2
#1 TRUE FALSE
#2 TRUE FALSE
#3 TRUE FALSE
```

see_diff

see_diff

Description

 $Output\ the\ opposite\ of\ intersect (a,b).\ Already\ seen\ at:\ https://stackoverflow.com/questions/19797954/function-to-find-symmetric-difference-opposite-of-intersection-in-r$

Usage

```
see\_diff(vec1 = c(), vec2 = c())
```

Arguments

vec1 is the first vector vec2 is the second vector

```
print(see_diff(c(1:7), c(4:12)))
[1] 1 2 3 8 9 10 11 12
```

see_diff_detailled

```
see_diff_all
```

see_diff_all

Description

Allow to perform the opposite of intersect function to n vectors.

Usage

```
see_diff_all(...)
```

Arguments

... are all the input vectors

Examples

```
vec1 <- c(3:6)
vec2 <- c(1:8)
vec3 <- c(12:16)

print(see_diff_all(vec1, vec2))

[1] 1 2 7 8

print(see_diff_all(vec1, vec2, vec3))

[1] 3 4 5 6 1 2 7 8 12 13 14 15 16</pre>
```

```
see_diff_detailled see_diff_detailled
```

Description

Behaves exactly like the see_diff function but is written more explicitely, see examples

Usage

```
see\_diff\_detailled(vec1 = c(), vec2 = c())
```

Arguments

```
vec1 is one of the input vector
vec2 is the other input vector
```

```
print(see_diff_detailled(c(1:6), c(3:9)))
[1] 1 2 7 8 9
```

see_file 141

see_file see_file

Description

Allow to get the filename or its extension

Usage

```
see_file(string_, index_ext = 1, ext = TRUE)
```

Arguments

string_ is the input string

index_ext is the occurence of the dot that separates the filename and its extension

ext is a boolean that if set to TRUE, will return the file extension and if set to FALSE,

will return filename

Examples

```
print(see_file(string_="file.abc.xyz"))
#[1] ".abc.xyz"
print(see_file(string_="file.abc.xyz", ext=FALSE))
#[1] "file"
print(see_file(string_="file.abc.xyz", index_ext=2))
#[1] ".xyz"
```

Description

Returns a boolean vector to see if a set of elements contained in v1 is also contained in another vector (v2)

Usage

```
see_idx(v1, v2)
```

Arguments

v1 is the first vector v2 is the second vector 142 see_inside

Examples

```
print(see_idx(v1=c("oui", "non", "peut", "oo"), v2=c("oui", "peut", "oui")))
#[1] TRUE FALSE TRUE FALSE
```

see_inside

see_inside

Description

Return a list containing all the column of the files in the current directory with a chosen file extension and its associated file and sheet if xlsx. For example if i have 2 files "out.csv" with 2 columns and "out.xlsx" with 1 column for its first sheet and 2 for its second one, the return will look like this: c(column_1, column_2, column_3, column_4, column_5, unique_separator, "1-2-out.csv", "3-3sheet_1-out.xlsx", 4-5-sheet_2-out.xlsx)

Usage

```
see_inside(
  pattern_,
  path_ = ".",
  sep_{-} = c(","),
  unique_sep = "#####",
  rec = FALSE
)
```

Arguments

is a vector containin the file extension of the spreadsheets ("xlsx", "csv"...) pattern_ is the path where are located the files path_ sep_ is a vector containing the separator for each csv type file in order following the operating system file order, if the vector does not match the number of the csv files found, it will assume the separator for the rest of the files is the same as the last csv file found. It means that if you know the separator is the same for all the csv type files, you just have to put the separator once in the vector. is a pattern that you know will never be in your input files unique_sep is a boolean allows to get files recursively if set to TRUE, defaults to TRUE If x rec is the return value, to see all the files name, position of the columns and possible

sheet name associanted with, do the following:

see_in_grep 143

see_in_grep	see_in_grep
0 0 0 <u></u>	500_m_5.0p

Description

Allow to get the indices of the elements of a vector that contains certyain patterns. The type of the output may change in function of the input vectors, see examples

Usage

```
see_in_grep(from_v = c(), in_v = c())
```

Arguments

is the vector that may contains elements that contains the same patterns that those in in_v, see examples
in_v is a vector that contains the patterns to find

Examples

```
see_in_l see_in_l
```

Description

Allow to get the patterns that are present in the elements of a vector, see examples

Usage

```
see_in_l(from_v = c(), in_v = c())
```

144 selected_char

Arguments

from_v is the vector that may contains elements that contains the same patterns that those in in_v, see examplesin_v is a vector that contains the patterns to find

Examples

```
print(see_in_l(from_v = c("oui", "non", "peut"),
   in_v = c("ou", "pe", "plm")))

  ou   pe   plm
  TRUE  TRUE  FALSE
```

see_mode

see_mode

Description

Allow to get the mode of a vector, see examples.

Usage

```
see\_mode(inpt\_v = c())
```

Arguments

inpt_v is the input vector

Examples

```
print(see_mode(inpt_v = c(1, 1, 2, 2, 2, 3, 1, 2)))
[1] 2
print(see_mode(inpt_v = c(1, 1, 2, 2, 2, 3, 1)))
[1] 1
```

selected_char

selected_char

Description

Allow to generate a char based on a conbinaison on characters from a vector and a number

Usage

```
selected_char(n, base_char = letters)
```

sequence_na_mean1 145

Arguments

```
n is how many sequence of numbers will be generated
base_char is the vector containing the elements from which the character is generated
```

Examples

```
print(selected_char(1222))
[1] "zta"
```

Description

In a dataframe generated by the function historic_sequence1, convert all NA to the mean of the values at the same variable for the individual at the id where the NA occurs, see examples (only accepts numeric variables)

Usage

```
sequence_na_mean1(inpt_datf, bf_)
```

Arguments

```
inpt_datf is the input dataframe
```

```
set.seed(123)
var1 < - round(runif(n = 14, min = 100, max = 122))
set.seed(123)
var2 \leftarrow round(runif(n = 14, min = 14, max = 20))
datf <- data.frame("ids" = c(20, 20, 20, 20, 19, 19, 19, 18, 18, 18, 18,
17, 17, 17),
"individual" = c("oui", "non", "peut1", "peut2",
"oui", "peut1", "peut2"),
"var1" = var1,
"var2" = var2)
datf <- historic_sequence1(inpt_datf = datf, bf_ = 2)</pre>
datf[3, 4] <- NA
datf[6, 4] \leftarrow NA
datf[1, 3] <- NA
print(datf)
  id_seq individual var1-1 var1-2 var2-1 var2-2
1
      20
                      NA
                            120
                                   20
                                             19
               oui
2
      20
                        NA
                              112
                                       NA
                                              17
               non
3
      20
              peut1
                       101
                               NA
                                       14
                                              17
      20
              peut2
                     112
                              121
                                      17
                                              20
5
      19
                oui
                      120
                              110
                                      19
                                              17
```

sequence_na_mean2

```
peut1
                      110
                            NA
                                    17
7
     19
             peut2
                      121
                            113
                                    20
                                           17
print(sequence_na_mean1(inpt_datf = datf, bf_ = 2))
 id_seq individual var1-1 var1-2 var2-1 var2-2
1
     2.0
                    115 120.0
                                  2.0
                                          19
              oui
                     112 112.0
2
     20
                                   17
                                          17
              non
                    101 105.5
3
     20
            peut1
                                   14
                                          17
                     112 121.0
4
     20
            peut2
                                   17
                                          20
5
     19
                     120 110.0
                                   19
                                          17
             oui
6
     19
            peut1
                     110 105.5
                                   17
                                          18
7
     19
             peut2
                    121 113.0
                                    20
                                          17
```

```
sequence_na_mean2 sequence_na_mean2
```

Description

In a dataframe generated by the function historic_sequence1, convert all NA to the mean of the values at the same variable for the individual at the id where the NA occurs, see examples (only accepts numeric variables)

Usage

```
sequence_na_mean2(inpt_datf, bf_)
```

Arguments

```
inpt_datf is the input dataframe

bf_ is how at how many n -1 we look for the value of the variables for the individual at time index n
```

```
set.seed(123)
var1 < - round(runif(n = 14, min = 100, max = 122))
set.seed(123)
var2 \leftarrow round(runif(n = 14, min = 14, max = 20))
datf <- data.frame("ids" = c(20, 20, 20, 20, 19, 19, 19, 18, 18, 18, 18,
17, 17, 17),
"individual" = c("oui", "non", "peut1", "peut2",
"oui", "peut1", "peut2"),
"var1" = var1,
"var2" = var2)
datf <- historic_sequence2(inpt_datf = datf, bf_ = 2)</pre>
datf[3, 4] <- NA
datf[6, 4] <- NA
datf[1, 3] <- NA
print(datf)
  id_seq individual var1-0 var1-1 var1-2 var2-0 var2-1 var2-2
```

sequence_na_med1 147

1	20	oui	NA	121	120	16	NA	19
2	20	non	117	NA	112	19	NA	17
3	20	peut1	109	NA	110	16	14	17
4	20	peut2	119	112	121	19	17	20
5	19	oui	121	120	110	20	19	17
6	19	peut1	101	NA	115	14	17	18
7	19	peut2	112	121	113	17	20	17
р	rint(sec	quence_na_me	ean2(inp	pt_datf =	datf, 1	$of_ = 2$))	
	id_seq	individual	var1-0	var1-1	var1-2	var2-0	var2-1	var2-2
1	20	oui	117	121.0000	120	16	18	19
2	20	non	117	114.5000	112	19	18	17
3	20	peut1	109	108.3333	110	16	14	17
4	20	peut2	119	112.0000	121	19	17	20
5	19	oui	121	120.0000	110	20	19	17
6	19	peut1	101	108.3333	115	14	17	18
7	19	peut2	112	121.0000	113	17	20	17

```
sequence_na_med1 sequence_na_med1
```

Description

In a dataframe generated by the function historic_sequence1, convert all NA to the median of the values at the same variable for the individual at the id where the NA occurs, see examples (only accepts numeric variables)

Usage

```
sequence_na_med1(inpt_datf, bf_)
```

Arguments

```
\begin{array}{ll} \text{inpt\_datf} & \text{is the input dataframe} \\ \text{bf\_} & \text{is how at how many } n \text{--}1 \text{ we look for the value of the variables for the individual} \\ & \text{at time index } n \end{array}
```

```
set.seed(123)
var1 <- round(runif(n = 14, min = 100, max = 122))
set.seed(123)
var2 <- round(runif(n = 14, min = 14, max = 20))

datf <- data.frame("ids" = c(20, 20, 20, 20, 19, 19, 19, 18, 18, 18, 18, 17, 17, 17),
   "individual" = c("oui", "non", "peut1", "peut2",
   "oui", "peut1", "peut2"),
   "var1" = var1,
   "var2" = var2)
datf <- historic_sequencel(inpt_datf = datf, bf_ = 2)
datf[3, 4] <- NA</pre>
```

148 sequence_na_med2

```
datf[6, 4] <- NA
datf[1, 3] \leftarrow NA
print(datf)
 id_seq individual var1-1 var1-2 var2-1 var2-2
1
           oui NA 120 20
    20
2
    20
                  NA
                      112
                              NA
            non
3
    20
         peut1
                 101
                        NA
                              14
                                    17
          peut2 112 121
4
    20
                              17
                                    20
5
           oui 120 110
    19
                              19
                                    17
                              17
         peut1
    19
                 110
                        NA
7
    19
           peut2
                121
                        113
                              20
                                    17
print(sequence_na_med1(inpt_datf = datf, bf_ = 2))
 id_seq individual var1-1 var1-2 var2-1 var2-2
    20 oui 115 120.0 20
1
                              17
2.
     2.0
       non
peut1
            non 112 112.0
                                     17
                              14
3
    20
                 101 105.5
                                     17
                  112 121.0
120 110.0
         peut2
4
    20
                               17
                                     20
5
    19
           oui
                               19
                                     17
        peut1
6
    19
                  110
                      105.5
                               17
                                     18
           peut2
                  121 113.0
7
    19
                               20
                                     17
```

```
sequence_na_med2 sequence_na_med2
```

Description

In a dataframe generated by the function historic_sequence2, convert all NA to the median of the values at the same variable for the individual at the id where the NA occurs, see examples (only accepts numeric variables)

Usage

```
sequence_na_med2(inpt_datf, bf_)
```

Arguments

```
inpt_datf is the input dataframe

bf_ is how at how many n -1 we look for the value of the variables for the individual at time index n
```

```
set.seed(123)
var1 <- round(runif(n = 14, min = 100, max = 122))
set.seed(123)
var2 <- round(runif(n = 14, min = 14, max = 20))
datf <- data.frame("ids" = c(20, 20, 20, 20, 19, 19, 19, 18, 18, 18, 17, 17, 17),
"individual" = c("oui", "non", "peut1", "peut2",
"oui", "peut1", "peut2"),</pre>
```

sort_date 149

```
"var1" = var1,
"var2" = var2)
datf <- historic_sequence2(inpt_datf = datf, bf_ = 2)</pre>
datf[3, 4] <- NA
datf[6, 4] <- NA
datf[1, 3] <- NA
print(datf)
 id_seg individual var1-0 var1-1 var1-2 var2-0 var2-1 var2-2
1
            oui NA 121 120 16 20 19
     20
     20
             non 117
                         NA 112
                                      19
                                             NA
3
     20
           peut1
                   109
                         NA 110
                                      16
                                            14
                                                  17
           peut2 119 112 121
4
     20
                                      19
                                            17
                                                  20
            oui 121 120 110 20 19
peut1 101 NA 115 14 17
5
     19
                                                  17
6
     19
                                                  18
            peut1
7
     19
                   112
                          121
                               113
                                      17
                                             20
                                                   17
           peut2
print(sequence_na_med2(inpt_datf = datf, bf_ = 2))
 id_seq individual var1-0 var1-1 var1-2 var2-0 var2-1 var2-2
     20
        oui 120 121.0 120
                                    16
                    117 114.5
109 109.0
     20
             non
                                112
                                       19
                                             18
3
     20
           peut1
                                110
                                       16
                                             14
                                                   17
                    119 112.0
4
     20
           peut2
                                121
                                       19
                                             17
                                                   20
                    121 120.0
                                110
115
                                      20
                                                   17
5
                                             19
     19
            oui
                    101 109.0
                                      14
                                                   18
6
     19
                                             17
            peut1
    19
                    112 121.0
                                113
7
                                      17
                                                  17
            peut2
                                             20
```

sort.	date	sort	date

Description

Allow to sort any vector containing a date, from any kind of format (my, hdmy, ymd ...), see examples.

Usage

```
sort_date(inpt_v, frmt, sep_ = "-", ascending = FALSE, give = "value")
```

Arguments

inpt_v	is the input vector containing all the dates
frmt	is the format of the dates, (any combinaison of letters "s" for second, "n", for minute, "h" for hour, "d" for day, "m" for month and "y" for year)
sep_	is the separator used for the dates
ascending	is the used to sort the dates
give	takes only two values "index" or "value", if give == "index", the function will output the index of sorted dates from inpt_v, if give == "value", the function will output the value, it means directly the sorted dates in inpt_v, see examples

Examples

```
sort_normal_qual sort_normal_qual
```

Description

Sort qualitative modalities that have their frequency normally distributed from an unordered dataset, see examples. This function uses an another algorythm than choose_normal_qual2 which may be faster.

Usage

```
sort_normal_qual(inpt_datf)
```

Arguments

inpt_datf is the input dataframe, containing the values in the first column and their frequency in the second

```
sample_val <- round(rnorm(n = 2000, mean = 12, sd = 2), 1)
sample_freq <- unique_total(sample_val)
sample_qual <- infinite_char_seq(n = length(sample_freq))
datf_test <- data.frame(sample_qual, sample_freq)
datf_test[, 2] <- datf_test[, 2] / sum(datf_test[, 2]) # optional</pre>
```

print(datf_test)

```
sample_qual sample_freq
            a 0.208695652
            b 0.234782609
             c 0.321739130
4
            d 0.339130435
            e 0.330434783
            f 0.069565217
7
            q 0.234782609
            h 0.400000000
9
            i 0.347826087
10
            j 0.043478261
11
            k 0.278260870
12
            1 0.286956522
13
           m 0.243478261
14
           n 0.147826087
15
            0 0.234782609
           p 0.252173913
16
           q 0.417391304
17
18
            r 0.095652174
19
             s 0.313043478
20
             t 0.008695652
21
             u 0.130434783
22
             v 0.391304348
23
            w 0.113043478
24
            x 0.295652174
25
            y 0.243478261
            z 0.382608696
26
          aa 0.008695652
27
          ab 0.347826087
28
29
          ac 0.330434783
30
          ad 0.321739130
31
          ae 0.347826087
32
          af 0.321739130
          ag 0.173913043
33
34
          ah 0.278260870
          ai 0.278260870
35
          aj 0.347826087
36
          ak 0.026086957
37
          al 0.295652174
38
          am 0.226086957
39
          an 0.295652174
40
           ao 0.234782609
41
42
            ap 0.113043478
43
            aq 0.234782609
44
            ar 0.173913043
45
            as 0.017391304
           at 0.252173913
46
47
           au 0.078260870
48
          av 0.086956522
49
          aw 0.278260870
50
          ax 0.086956522
51
          ay 0.200000000
52
          az 0.295652174
53
          ba 0.052173913
54
          bb 0.165217391
```

55	bc	0.408695652
56	bd	0.269565217
57	be	0.104347826
58	bf	0.391304348
59	bg	0.104347826 0.043478261
60 61	bh b:	0.200000000
62	bi bj	0.095652174
63	bk	0.191304348
64	bl	0.008695652
65	bm	0.165217391
66	bn	0.226086957
67	bo	0.086956522
68	bp	0.017391304
69	bq	0.121739130
70	br	0.234782609
71	bs	0.121739130
72	bt	0.078260870
73	bu	0.173913043
74	bv	0.104347826
75	bw	0.208695652
76	bx	0.017391304
77	рÀ	0.243478261
78	bz	0.034782609
79	ca	0.017391304
80	cb	0.008695652
81	CC	0.173913043
82	cd	0.147826087
83 84	ce	0.060869565 0.017391304
85	cf	0.060869565
86	cg ch	0.008695652
87	ci	0.208695652
88	сj	0.043478261
89	ck	0.052173913
90	cl	0.017391304
91	cm	0.017391304
92	cn	0.095652174
93	СО	0.113043478
94	ср	0.017391304
95	cq	0.017391304
96	cr	0.026086957
97	CS	0.034782609
98	ct	0.017391304
99	cu	0.026086957
100	CV	0.026086957
101	CW	0.026086957 0.017391304
102 103	CX	0.017391304
104	СŻ	0.008695652
105	da	0.034782609
106	db	0.034782009
107	dc	0.060869565
108	dd	0.008695652
109	de	0.008695652
110	df	0.017391304
111	dg	0.008695652
	_	

112

```
dh 0.008695652
113
             di 0.017391304
114
             dj 0.008695652
             dk 0.008695652
115
print(sort_normal_qual(inpt_datf = datf_test))
0.00869565217391304 0.00869565217391304 0.00869565217391304 0.00869565217391304
              "aa"
                                   "cb"
                                                         "cz"
0.00869565217391304 \ 0.00869565217391304 \ 0.0173913043478261 \ 0.0173913043478261
              "dh"
                                   "dk"
                                                         "bp"
                                                                              "ca"
0.0173913043478261
                    0.0173913043478261
                                          0.0173913043478261
                                                               0.0173913043478261
              "cl"
                                   "ср"
                                                         "ct"
0.0173913043478261
                     0.0260869565217391
                                          0.0260869565217391
                                                               0.0347826086956522
              "di"
                                   "cr"
                                                         "cv"
                                                                              "bz"
0.0347826086956522
                                          0.0434782608695652
                    0.0434782608695652
                                                               0.0521739130434783
              "da"
                                   "bh"
                                                         "cy"
                                                                              "ck"
0.0608695652173913
                    0.0695652173913043
                                          0.0782608695652174
                                                               0.0869565217391304
              "cq"
                                     "f"
                                                         "bt"
                                                                              "ax"
0.0956521739130435
                     0.0956521739130435
                                           0.104347826086957
                                                                 0.11304347826087
               "r"
                                    "cn"
                                                         "bq"
  0.11304347826087
                      0.121739130434783
                                           0.147826086956522
                                                                0.165217391304348
               "co"
                                    "bs"
                                                          "n"
                                                                              "bb"
 0.173913043478261
                      0.173913043478261
                                           0.191304347826087
                                                                               0.2
                                                                              "bi"
              "ag"
                                    "bu"
                                                         "bk"
 0.208695652173913
                      0.226086956521739
                                           0.234782608695652
                                                                0.234782608695652
              "bw"
                                    "am"
                                                          "b"
                                                                               " 0 "
                                           0.243478260869565
 0.234782608695652
                      0.243478260869565
                                                                0.252173913043478
               "aq"
                                     "m"
                                                         "bv"
                                                                              "at"
 0.278260869565217
                      0.278260869565217
                                            0.28695652173913
                                                                0.295652173913043
                " k "
                                                          " | "
                                    "ai"
                                                                              "al"
 0.295652173913043
                                           0.321739130434783
                                                                0.330434782608696
                      0.321739130434783
               "az"
                                     "c"
                                                         "af"
                                                                              "ac"
 0.347826086956522
                      0.347826086956522
                                           0.382608695652174
                                                                0.391304347826087
               " i "
                                    "ae"
                                                          "z"
                                                                              "bf"
 0.408695652173913
                      0.417391304347826
                                                          0.4
                                                                0.391304347826087
                                     "q"
                                                          "h"
               "ho"
                                                                               11 77 11
 0.347826086956522
                      0.347826086956522
                                           0.339130434782609
                                                                0.330434782608696
               "aj"
                                    "ab"
                                                          "d"
                                                                               "e"
 0.321739130434783
                       0.31304347826087
                                           0.295652173913043
                                                                0.295652173913043
                                     "s"
               "ad"
                                                         "an"
                                                                               " × "
                      0.278260869565217
 0.278260869565217
                                           0.269565217391304
                                                                0.252173913043478
               "aw"
                                    "ah"
                                                         "bd"
 0.243478260869565
                      0.234782608695652
                                           0.234782608695652
                                                                0.234782608695652
                                   "br"
                                                         "ao"
                                                                               "g"
 0.226086956521739
                                                                               0.2
                      0.208695652173913
                                           0.208695652173913
              "bn"
                                   "ci"
                                                         "a"
                                                                              "ay"
 0.173913043478261
                      0.173913043478261
                                           0.165217391304348
                                                                0.147826086956522
              "cc"
                                   "ar"
                                                         "bm"
                                                                              "cd"
 0.130434782608696
                                            0.11304347826087
                                                                0.104347826086957
                      0.121739130434783
               "u"
                                   "ba"
                                                         "ap"
                                                                              "bv"
 0.104347826086957
                     0.0956521739130435
                                          0.0869565217391304
                                                               0.0869565217391304
              "be"
                                   "bj"
                                                         "bo"
0.0782608695652174
                     0.0608695652173913
                                          0.0608695652173913
                                                               0.0521739130434783
              "au"
                                   "dc"
                                                        "ce"
                     0.0434782608695652
0.0434782608695652
                                          0.0347826086956522
                                                               0.0260869565217391
                                    "j"
              "cj"
                                                         "cs"
                                                                              "CW"
```

```
0.0260869565217391 0.0260869565217391 0.0173913043478261 0.0173913043478261
               "cu"
                                    "ak"
                                                          "df"
0.0173913043478261 \quad 0.0173913043478261 \quad 0.0173913043478261 \quad 0.0173913043478261
              "cq"
                                    "cm"
                                                          "cf"
0.0173913043478261 \ 0.00869565217391304 \ 0.00869565217391304 \ 0.00869565217391304
              "as"
                                    "dj"
                                                          "dq"
                                                                               "dd"
0.00869565217391304 0.00869565217391304 0.00869565217391304
               "ch"
                                    "bl"
```

```
sort_normal_qual2 sort_normal_qual2
```

Description

Sort qualitative modalities that have their frequency normally distributed from an unordered dataset, see examples. This function uses an another algorythm than choose_normal_qual which may be faster.

Usage

```
sort_normal_qual2(inpt_datf)
```

Arguments

inpt_datf is the input dataframe, containing the values in the first column and their frequency in the second

```
sample_val \leftarrow round(rnorm(n = 2000, mean = 12, sd = 2), 1)
sample_freq <- unique_total(sample_val)</pre>
sample_qual <- infinite_char_seq(n = length(sample_freq))</pre>
datf_test <- data.frame(sample_qual, sample_freq)</pre>
datf_test[, 2] <- datf_test[, 2] / sum(datf_test[, 2])</pre>
print(datf_test)
   sample_qual sample_freq
1
              a 0.208695652
               b 0.234782609
2
3
               c 0.321739130
4
               d 0.339130435
5
               e 0.330434783
6
               f 0.069565217
7
               g 0.234782609
8
               h 0.40000000
9
               i 0.347826087
1.0
               j 0.043478261
11
               k 0.278260870
12
              1 0.286956522
13
              m 0.243478261
              n 0.147826087
15
               o 0.234782609
```

16	р	0.252173913
17	_	0.417391304
	q	
18	r	0.095652174
19	S	0.313043478
20	t	0.008695652
21	u	0.130434783
22	V	0.391304348
23	W	0.113043478
24		0.295652174
	Х	
25	У	0.243478261
26	Z	0.382608696
27	aa	0.008695652
28	ab	0.347826087
29	ac	0.330434783
30	ad	0.321739130
31	ae	0.347826087
32	af	0.321739130
33	ag	
34	ah	0.278260870
35	ai	0.278260870
36	аj	0.347826087
37	ak	0.026086957
38	al	0.295652174
39		0.226086957
	am	
40	an	0.295652174
41	ao	0.234782609
42	ap	0.113043478
43	aq	0.234782609
44	ar	0.173913043
45	as	0.017391304
46		0.252173913
	at	
47	au	0.078260870
48	av	0.086956522
49	aw	0.278260870
50	ax	0.086956522
51	ay	0.200000000
52	az	0.295652174
53	ba	0.052173913
54		0.165217391
	bb	
55	bc	0.408695652
56	bd	0.269565217
57	be	0.104347826
58	bf	0.391304348
59	bg	0.104347826
60	bh	0.043478261
61	bi	0.200000000
		0.095652174
62	bj	
63	bk	0.191304348
64	bl	0.008695652
65	bm	0.165217391
66	bn	0.226086957
67	bo	0.086956522
68	bp	0.017391304
	_	
69	bq	0.121739130
70	br	0.234782609
71	bs	0.121739130
72	bt	0.078260870

```
73
             bu 0.173913043
74
             bv 0.104347826
75
             bw 0.208695652
76
             bx 0.017391304
77
             by 0.243478261
78
             bz 0.034782609
79
             ca 0.017391304
             cb 0.008695652
80
             cc 0.173913043
81
82
            cd 0.147826087
83
            ce 0.060869565
            cf 0.017391304
85
            cg 0.060869565
86
            ch 0.008695652
87
            ci 0.208695652
88
            cj 0.043478261
89
             ck 0.052173913
90
             cl 0.017391304
91
             cm 0.017391304
92
             cn 0.095652174
93
             co 0.113043478
94
             ср 0.017391304
95
             cq 0.017391304
96
             cr 0.026086957
97
             cs 0.034782609
98
             ct 0.017391304
             cu 0.026086957
99
             cv 0.026086957
100
             cw 0.026086957
101
102
            cx 0.017391304
103
            cy 0.043478261
            cz 0.008695652
104
105
            da 0.034782609
106
            db 0.017391304
107
            dc 0.060869565
108
            dd 0.008695652
109
            de 0.008695652
             df 0.017391304
110
             dg 0.008695652
111
             dh 0.008695652
112
113
             di 0.017391304
114
             dj 0.008695652
115
             dk 0.008695652
print(sort_normal_qual2(inpt_datf = datf_test))
0.00869565217391304 \ 0.00869565217391304 \ 0.00869565217391304 \ 0.00869565217391304
              "aa"
                                   "cb"
                                                         "cz"
0.00869565217391304 \ \ 0.00869565217391304 \ \ \ 0.0173913043478261 \ \ \ 0.0173913043478261
               "dh"
                                    "dk"
                                                         "bp"
0.0173913043478261 \quad 0.0173913043478261 \quad 0.0173913043478261 \quad 0.0173913043478261
              "cl"
                                    "cp"
                                                         "ct"
0.0173913043478261 \quad 0.0260869565217391 \quad 0.0260869565217391 \quad 0.0347826086956522
              "di"
                                   "cr"
                                                         "cv"
0.0347826086956522 \quad 0.0434782608695652 \quad 0.0434782608695652 \quad 0.0521739130434783
              "da"
                                   "bh"
                                                         "cy"
```

 $0.0608695652173913 \quad 0.0695652173913043 \quad 0.0782608695652174 \quad 0.0869565217391304$

split_by_step 157

"cg"	"f"	"bt"	"ax"
0.0956521739130435	0.0956521739130435	0.104347826086957	0.11304347826087
"r"	"cn"	"bg"	" _W "
0.11304347826087	0.121739130434783	0.147826086956522	0.165217391304348
"co"	"bs"	"n"	"bb"
0.173913043478261	0.173913043478261	0.191304347826087	0.2
"ag"	"bu"	"bk"	"bi"
0.208695652173913	0.226086956521739	0.234782608695652	0.234782608695652
"bw"	"am"	"b"	"0"
0.234782608695652	0.243478260869565	0.243478260869565	0.252173913043478
"aq"	"m"	"by"	"at"
0.278260869565217	0.278260869565217	0.28695652173913	0.295652173913043
"k"	"ai"	"1"	"al"
0.295652173913043	0.321739130434783	0.321739130434783	0.330434782608696
"az"	" _C "	"af"	"ac"
0.347826086956522	0.347826086956522	0.382608695652174	0.391304347826087
"i"	"ae"	" _Z "	"bf"
0.408695652173913	0.417391304347826	0.4	0.391304347826087
"bc"	"q"	"h"	"V"
0.347826086956522	0.347826086956522	0.339130434782609	0.330434782608696
"aj"	"ab"	"d"	"e"
0.321739130434783	0.31304347826087	0.295652173913043	0.295652173913043
"ad"	"s"	"an"	" _X "
0.278260869565217	0.278260869565217	0.269565217391304	0.252173913043478
"aw"	"ah"	"bd"	"p"
0.243478260869565	0.234782608695652	0.234782608695652	0.234782608695652
" _Y "	"br"	"ao"	" g"
0.226086956521739	0.208695652173913	0.208695652173913	0.2
"bn"	"ci"	"a"	"ay"
0.173913043478261	0.173913043478261	0.165217391304348	0.147826086956522
"cc"	"ar"	"bm"	"cd"
0.130434782608696	0.121739130434783	0.11304347826087	0.104347826086957
"u"	"bq"	"ap"	"bv"
0.104347826086957	0.0956521739130435	0.0869565217391304	0.0869565217391304
"be"	"bj"	"bo"	"av"
0.0782608695652174	0.0608695652173913	0.0608695652173913	0.0521739130434783
"au"	"dc"	"ce"	"ba"
0.0434782608695652	0.0434782608695652	0.0347826086956522	0.0260869565217391
" cj"	" j"	"cs"	"cw"
0.0260869565217391	0.0260869565217391	0.0173913043478261	0.0173913043478261
"cu"	"ak"	"df"	"CX"
0.0173913043478261	0.0173913043478261	0.0173913043478261	0.0173913043478261
"cq"	"cm"	"cf"	"bx"
0.0173913043478261	0.00869565217391304	0.00869565217391304	0.00869565217391304
"as"	"dj"	"dg"	"dd"
0.00869565217391304	0.00869565217391304	4 0.00869565217391304	1
"ch"	"bl"	"t"	

split_by_step

Description

Allow to split a string or a vector of strings by a step, see examples.

158 str_remove_untl

Usage

```
split_by_step(inpt_v, by)
```

Arguments

```
inpt_v is the input character or vector of characters
by is the step
```

Examples

```
print(split_by_step(inpt_v = c("o", "u", "i", "n", "o", "o", "u", "i", "o", "Z"), by = 2)
[1] "ou" "in" "oo" "ui" "oZ"

print(split_by_step(inpt_v = c("o", "u", "i", "n", "o", "o", "u", "i", "o", "Z"), by = 3)
[1] "oui" "noo" "uio" "Z"

print(split_by_step(inpt_v = c("o", "u", "i", "n", "o", "o", "u", "i", "o", "Z"), by = 4)
[1] "ouin" "ooui" "oZ"

print(split_by_step(inpt_v = 'ouinoouioz', by = 4))
[1] "ouin" "ooui" "oZ"
```

```
str_remove_untl str_remove_untl
```

Description

Allow to remove pattern within elements from a vector precisely according to their occurence.

Usage

```
str_remove_untl(
  inpt_v,
  ptrn_rm_v = c(),
  untl = list(c(1)),
  nvr_following_ptrn = "NA"
)
```

Arguments

sub_mult 159

Examples

```
vec <- c("45/56-/98mm", "45/56-/98mm", "45/56-/98-mm//")
print(str_remove_untl(inpt_v=vec, ptrn_rm_v=c("-", "/"), untl=list(c("max"), c(1))))
#[1] "4556/98mm" "4556/98mm" "4556/98mm//"
print(str_remove_untl(inpt_v=vec, ptrn_rm_v=c("-", "/"), untl=list(c("max"), c(1:2))))
#[1] "455698mm" "455698mm" "455698mm//"
print(str_remove_untl(inpt_v=vec[1], ptrn_rm_v=c("-", "/"), untl=c("max")))
#[1] "455698mm" "455698mm" "455698mm"</pre>
```

sub_mult

sub_mult

Description

Performs a sub operation with n patterns and replacements.

Usage

```
sub_mult(inpt_v, pattern_v = c(), replacement_v = c())
```

Arguments

inpt_v is a vector containing all the elements that contains expressions to be substituted
pattern_v is a vector containing all the patterns to be substituted in any elements of inpt_v
replacement_v

is a vector containing the expression that are going to substituate those provided by pattern_v

sum_group1

```
successive_diff successive_diff
```

Description

Allow to see the difference beteen the suxxessive elements of an numeric vector

Usage

```
successive_diff(inpt_v)
```

Arguments

```
inpt_v is the input numeric vector
```

Examples

```
print(successive_diff(c(1:10)))
[1] 1 1 1 1 1
print(successive_diff(c(1:11, 13, 19)))
[1] 1 1 1 1 2 6
```

```
sum_group1
```

sum_group1

Description

Allow to aggregate variables according to groups, do not visually group the individual unlike sum_group2, see examples

Usage

```
sum_group1(inpt_datf, col_grp = c(), col_to_add = c())
```

Arguments

```
inpt_datf is the input dataframe
col_grp is a vector containing the column names or the column numbers of the groups
col_to_add is a vector containing the column names or the column numbers of the variables to aggregate
```

sum_group1 161

```
set.seed(123)
datf <- data.frame("country" = c("France", "Germany", "France", "Italy", "Italy", "France"
                  "year" = c(2012, 2012, 2013, 2011, 2012, 2011),
                  "comp_arm" = c("higher", "lower", "higher", "lower", "lower"
                  "pop" = runif(n = 6, min = 65000000, max = 69000000),
                  "random_var" = round(x = runif(n = 6, min = 16, max = 78), digits = 0)
datf
 country year comp_arm
                          pop random_var
1 France 2012 higher 66150310
2 Germany 2012 lower 68153221
3 France 2013 higher 66635908
                                      50
  Italy 2011 higher 68532070
                                      44
  Italy 2012 lower 68761869
                                      75
6 France 2011 lower 65182226
                                      44
print(sum_group1(inpt_datf = datf, col_grp = c("country", "year"), col_to_add = c("random
                           pop random_var
  country year comp_arm
1 France 2012 higher 66150310
2 Germany 2012
                lower 68153221
               higher 66635908
3 France 2013
                                      50
               higher 68532070
   Italy 2011
                                       44
               lower 68761869
   Italy 2012
                                       75
                lower 65182226
6 France 2011
                                       44
print(sum_group1(inpt_datf = datf, col_grp = c("year"), col_to_add = c("random_var", "por
  country year comp_arm
                            pop random_var
1 France 2012 higher 203065400
                                     195
2 Germany 2012 lower 203065400
                                      195
3 France 2013 higher 66635908
                                       50
  Italy 2011 higher 133714296
  Italy 2012 lower 203065400
                                      195
6 France 2011 lower 133714296
                                       88
print(sum_group1(inpt_datf = datf, col_grp = c("country"), col_to_add = c("random_var", '
 country year comp_arm
                           pop random_var
1 France 2012 higher 197968444 143
2 Germany 2012
                lower 68153221
                                       71
3 France 2013 higher 197968444
                                       143
              higher 137293939
   Italy 2011
                                       119
               lower 137293939
   Italy 2012
                                       119
6 France 2011
                lower 197968444
                                       143
set.seed(123)
pop_v \leftarrow runif(n = 6, min = 65000000, max = 69000000)
pop_v[c(1, 3)] <- NA
set.seed(123)
datf <- data.frame("country" = c("France", "Germany", "France", "Italy", "Italy", "France")</pre>
                  "year" = c(2012, 2012, 2013, 2011, 2012, 2011),
                  "comp_arm" = c("higher", "lower", "higher", "lower", "lower"
                  "pop" = pop_v,
                  "random_var" = round(x = runif(n = 6, min = 16, max = 78), digits = 0)
```

162 sum_group2

```
datf
```

```
country year comp_arm pop random_var 1 France 2012 higher NA 34
2 Germany 2012 lower 68153221
                                    65
3 France 2013 higher NA
                                    41
   Italy 2011 higher 68532070
                                    71
  Italy 2012 lower 68761869
                                    74
6 France 2011 lower 65182226
                                    19
print(sum_group1(inpt_datf = datf, col_grp = c("year"), col_to_add = c("random_var", "por
 country year comp_arm
                          pop random_var
1 France 2012 higher 136915090 173
2 Germany 2012 lower 136915090
                                    173
3 France 2013 higher NA
                                    41
4 Italy 2011 higher 133714296
                                     90
5
  Italy 2012 lower 136915090
                                   173
6 France 2011 lower 133714296
```

3 France 2013 higher 66635908

Description

Allow to aggregate variables according to groups, see examples

Usage

```
sum_group2(inpt_datf, col_grp = c(), col_to_add = c())
```

Arguments

inpt_datf is the input dataframe

col_grp is a vector containing the column names or the column numbers of the groups

col_to_add is a vector containing the column names or the column numbers of the variables to aggregate

Examples

50

sum_group2 163

higher 68532070

Italy 2011

```
lower 68761869
   Italy 2012
                                      75
6 France 2011
                lower 65182226
                                      44
print(sum_group2(inpt_datf = datf, col_grp = c("country"), col_to_add = c("random_var", '
 country year comp_arm
                          pop random_var
1 France 2012 higher 197968444 143
3 France 2013 higher 197968444
                                      143
6 France 2011 lower 197968444
                                      143
2 Germany 2012 lower 68153221
                                       71
4 Italy 2011 higher 137293939
                                      119
5 Italy 2012 lower 137293939
                                      119
print(sum_group2(inpt_datf = datf, col_grp = c("year"), col_to_add = c("random_var", "por
 country year comp_arm
                           pop random_var
1 France 2012 higher 203065400 195
2 Germany 2012 lower 203065400
                                      195
  Italy 2012
                lower 203065400
                                     195
3 France 2013 higher 66635908
   Italy 2011 higher 133714296
6 France 2011 lower 133714296
print(sum_group2(inpt_datf = datf, col_grp = c("country", "year"), col_to_add = c("random
 country year comp_arm
                          pop random_var
1 France 2012 higher 66150310
2 Germany 2012
               lower 68153221
                                      71
3 France 2013 higher 66635908
4 Italy 2011 higher 68532070
                                     44
  Italy 2012 lower 68761869
                                      75
6 France 2011 lower 65182226
                                     44
set.seed(123)
pop_v \leftarrow runif(n = 6, min = 65000000, max = 69000000)
pop_v[c(1, 3)] <- NA
set.seed(123)
datf <- data.frame("country" = c("France", "Germany", "France", "Italy", "Italy", "France")</pre>
                  "year" = c(2012, 2012, 2013, 2011, 2012, 2011),
                  "comp_arm" = c("higher", "lower", "higher", "higher", "lower", "lower"
                  "pop" = pop_v,
                  "random_var" = round(x = runif(n = 6, min = 16, max = 78), digits = 0)
 country year comp_arm pop random_var
France 2012 higher NA 34
               lower 68153221
                                      65
2 Germany 2012
3 France 2013 higher NA
                                      41
  Italy 2011 higher 68532070
                                      71
  Italy 2012 lower 68761869
                                      74
6 France 2011 lower 65182226
                                      19
print(sum_group2(inpt_datf = datf, col_grp = c("year"), col_to_add = c("random_var", "por
 country year comp_arm pop random_var
1 France 2012 higher 136915090 173
2 Germany 2012 lower 136915090
                                     173
```

swipr

```
5 Italy 2012 lower 136915090 173
3 France 2013 higher NA 41
4 Italy 2011 higher 133714296 90
6 France 2011 lower 133714296 90
```

Description

Returns an ordered dataframes according to the elements order given. The input datafram has two columns, one with the ids which can be bonded to multiple elements in the other column.

Usage

```
swipr(inpt_datf, how_to = c(), id_w = 2, id_ids = 1)
```

Arguments

```
    inpt_datf
    is the input dataframe
    how_to
    is a vector containing the elements in the order wanted
    id_w
    is the column number or the column name of the elements
    id_ids
    is the column number or the column name of the ids
```

Examples

7

8

9

10

Arg

Arm

Arm

Al

В

G

В

```
datf <- data.frame("col1"=c("Af", "Al", "Al", "Al", "Arg", "Arg", "Arg", "Arm", "Arm",
                                                                       "col2"=c("B", "B", "G", "S", "B", "S", "G", "B", "G", "B"))
print(swipr(inpt_datf=datf, how_to=c("G", "S", "B")))
                          col1 col2
 1
                                         Αf
                                                                                    В
 2
                                         Al
                                                                                               G
 3
                                         Al
                                                                                               S
 4
                                       Al
                                                                                             В
 5
                                  Arg
                                                                                            G
 6
                                  Arg
                                                                                           S
```

test_order 165

test_order

 $test_order$

Description

Allow to get if two vectors have their commun elements in the same order, see examples

Usage

```
test_order(inpt_v_from, inpt_v_test)
```

Arguments

is

the vector we want to test if its commun element with inpt_v_from are in the same order

Examples

```
print(test_order(inpt_v_from = c(1:8), inpt_v_test = c(1, 4)))
[1] TRUE
print(test_order(inpt_v_from = c(1:8), inpt_v_test = c(1, 4, 2)))
[1] FALSE
```

Description

Allow ewualize the occurence of each elements in all the timestamps, see examples

Usage

```
time_serie_equalizer(
  inpt_datf,
  time_col,
  null_value = 0,
  individual_col,
  var_col = c()
)
```

166 time_serie_equalizer

Arguments

Examples

print(datf)

```
individual country year
                                                   energy_source twh_cons
1
           A France 1995
                                             biofuel_electricity
                                                                   1.82
2
           A France 1996
                                               coal_electricity
                                                                    24.18
           A France 1997
3
                                                 gas_electricity
                                                                    3.84
           A France 1998
                                               hydro_electricity
4
                                                                   71.33
           A France 1999
                                                                  377.23
5
                                             nuclear_electricity
6
           A France 2000
                                                 oil_electricity
                                                                   10.50
           A France 2001 other_renewable_exc_biofuel_electricity
                                                                    0.51
10
           B France 1995
                                             biofuel_electricity
                                                                    9.50
11
           B France 1996
                                                coal_electricity
                                                                    2.16
12
           B France 1997
                                                 gas_electricity
                                                                    31.43
           B France 1998
13
                                               hydro_electricity
                                                                   53.19
14
           B France 1999
                                             nuclear_electricity
                                                                   335.65
15
          B France 2000
                                                 oil_electricity
                                                                    9.71
          B France 2001 other_renewable_exc_biofuel_electricity
16
                                                                    0.60
                                               solar_electricity
17
          B France 2002
                                                                    23.26
18
          B France 2003
                                                wind_electricity
                                                                    48.61
```

	individual	country	year	energy_source	twh_cons
1	A	France	1995	biofuel_electricity	1.82
2	A	France	1996	coal_electricity	24.18
3	A	France	1997	gas_electricity	3.84
4	A	France	1998	hydro_electricity	71.33
5	A	France	1999	nuclear_electricity	377.23
6	A	France	2000	oil_electricity	10.50
7	A	France	2001	other_renewable_exc_biofuel_electricity	0.51
8	В	France	1995	biofuel_electricity	9.50
9	В	France	1996	coal_electricity	2.16
10	В	France	1997	gas_electricity	31.43
11	В	France	1998	hydro_electricity	53.19
12	В	France	1999	nuclear_electricity	335.65
13	В	France	2000	oil_electricity	9.71
14	В	France	2001	other_renewable_exc_biofuel_electricity	0.60
15	В	France	2002	solar_electricity	23.26
16	В	France	2003	wind_electricity	48.61
17	A	France	2002	biofuel_electricity	0.00

to_unique 167

18 A France 2003

biofuel_electricity 0.00

to_unique to_unique

Description

Allow to transform a vector containing elements that have more than 1 occurrence to a vector with only uniques elements.

Usage

```
to_unique(inpt_v, distinct_type = "suffix", distinct_val = "number", sep = "-")
```

Arguments

distinct_val takes two values: number (unique sequence of number to differencfiate each value) or letter (unique sequence of letters to differenciate each value)

```
print(to_unique(inpt_v = c("a", "a", "e", "a", "i", "i"),
                distinct_type = "suffix",
                distinct_val = "number",
                sep = "-"))
[1] "a-1" "a-2" "e" "a-3" "i-1" "i-2"
print(to_unique(inpt_v = c("a", "a", "e", "a", "i", "i"),
                distinct_type = "suffix",
                distinct_val = "letter",
                sep = "-"))
[1] "a-a" "a-b" "e"
                      "a-c" "i-a" "i-b"
print(to_unique(inpt_v = c("a", "a", "e", "a", "i", "i"),
                distinct_type = "prefix",
                distinct_val = "number",
                sep = "/"))
[1] "1/a" "2/a" "e" "3/a" "1/i" "2/i"
print(to_unique(inpt_v = c("a", "a", "e", "a", "i", "i"),
                distinct_type = "prefix",
                distinct_val = "letter",
                sep = "_"))
[1] "a_a" "b_a" "e" "c_a" "a_i" "b_i"
```

union_keep

union_all

union_all

Description

Allow to perform a union function to n vectors.

Usage

```
union_all(...)
```

Arguments

... are all the input vectors

Examples

```
print(union_all(c(1, 2), c(3, 4), c(1:8)))
[1] 1 2 3 4 5 6 7 8
print(union_all(c(1, 2), c(3, 4), c(7:8)))
[1] 1 2 3 4 7 8
```

union_keep

union_keep

Description

Performs a union operation keeping the number of elements of all input vectors, see examples

Usage

```
union_keep(...)
```

Arguments

... are all the input vectors

```
print(union_keep(c("a", "ee", "ee"), c("p", "p", "a", "i"), c("a", "a", "z")))
[1] "a" "ee" "ee" "p" "p" "i" "z"
print(union_keep(c("a", "ee", "ee"), c("p", "p", "a", "i")))
[1] "a" "ee" "ee" "p" "p" "i"
```

unique_datf 169

unique_	_datf	unique_	_datf

Description

Returns the input dataframe with the unique columns or rows.

Usage

```
unique_datf(inpt_datf, col = FALSE)
```

Arguments

```
inpt_datf is the input dataframe
col is a parameter that specifies if the dataframe returned should have unique columns
or rows, defaults to F, so the dataframe returned by default has unique rows
```

```
datf1 <- data.frame(c(1, 2, 1, 3), c("a", "z", "a", "p"))</pre>
print(datf1)
  c.1..2..1..3. c..a...z...a...p.. c.1..2..1..3..1
1
           1
                                а
             2
3
             1
             3
print (unique_datf(inpt_datf=datf1))
   c.1..2..1..3. c..a...z...a...p..
#1
         1
#2
              2
datf1 \leftarrow data.frame(c(1, 2, 1, 3), c("a", "z", "a", "p"), c(1, 2, 1, 3))
print(datf1)
  c.1..2..1..3. c..a...z...a...p..
1
       1
2
             2
3
             1
                                  а
print(unique_datf(inpt_datf=datf1, col=TRUE))
# cur_v cur_v
#1 1 a
     2
#2
#3
      1
#4
      3
```

170 unique_pos

```
unique_ltr_from_v
```

Description

Returns the unique characters contained in all the elements from an input vector "inpt_v"

Usage

```
unique_ltr_from_v(inpt_v, keep_v = c("?", "!", ":", "&", ",", ".", letters))
```

Arguments

```
inpt_v is the input vector containing all the elements
```

keep_v is the vector containing all the characters that the elements in inpt_v may contain

Examples

```
print(unique_ltr_from_v(inpt_v=c("bonjour", "lpoerc", "nonnour", "bonnour", "nonjour", "a
#[1] "b" "o" "n" "j" "u" "r" "l" "p" "e" "c" "a" "v" "i"
```

unique_pos

unique_pos

Description

Allow to find the first index of the unique values from a vector.

Usage

```
unique_pos(vec)
```

Arguments

vec

is the input vector

```
print(unique_pos(vec=c(3, 4, 3, 5, 6)))
#[1] 1 2 4 5
```

unique_total 171

unique_total unique_total

Description

Returns a vector with the total amount of occurences for each element in the input vector. The occurences of each element follow the same order as the unique function does, see examples

Usage

```
unique_total(inpt_v = c())
```

Arguments

inpt_v

is the input vector containing all the elements

Examples

```
print (unique_total(inpt_v = c(1:12, 1)))

[1] 2 1 1 1 1 1 1 1 1 1 1 1

print (unique_total(inpt_v = c(1:12, 1, 11, 11)))

[1] 2 1 1 1 1 1 1 1 1 3 1

vec <- c(1:12, 1, 11, 11)

names (vec) <- c(1:15)
print (unique_total(inpt_v = vec))

1 2 3 4 5 6 7 8 9 10 11 12
2 1 1 1 1 1 1 1 1 3 1</pre>
```

until_stnl

 $until_stnl$

Description

Maxes a vector to a chosen length. ex: if i want my vector c(1, 2) to be 5 of length this function will return me: c(1, 2, 1, 2, 1)

Usage

```
until_stnl(vec1, goal)
```

Arguments

vec1 is the input vector goal is the length to reach

172 vector_replacor

Examples

```
print(until_stnl(vec1=c(1, 3, 2), goal=56))
# [1] 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2
```

val_replacer

val_replacer

Description

Allow to replace value from dataframe to another one.

Usage

```
val_replacer(datf, val_replaced, val_replacor = TRUE)
```

Arguments

```
datf is the input dataframe

val_replaced is a vector of the value(s) to be replaced

val_replacor is the value that will replace val_replaced
```

Examples

```
# C.1...004..T.F. C.1..F. NA
#1 1 NA
#2 004 FALSE
#3 NA NA
#4 FALSE NA
```

vector_replacor

vector_replacor

Description

Allow to replace certain values in a vector.

Usage

```
vector_replacor(inpt_v = c(), sus_val = c(), rpl_val = c(), grep_ = FALSE)
```

vec_in_datf

Arguments

inpt_v	is the input vector
sus_val	is a vector containing all the values that will be replaced
rpl_val	is a vector containing the value of the elements to be replaced (sus_val), so sus_val and rpl_val should be the same size
grep_	is if the elements in sus_val should be equal to the elements to replace in inpt_v or if they just should found in the elements

Examples

vec_in_datf vec_in_datf

Description

Allow to get if a vector is in a dataframe. Returns the row and column of the vector in the dataframe if the vector is contained in the dataframe.

Usage

```
vec_in_datf(
  inpt_datf,
  inpt_vec = c(),
  coeff = 0,
  stop_untl = 1,
  conventional = FALSE
)
```

Arguments

inpt_datf is the input dataframe
inpt_vec is the vector that may be in the input dataframe
coeff is the "slope coefficient" of inpt_vec
stop_untl is the maximum number of the input vector the function returns, if in the dataframe
conventional is if a positive slope coefficient means that the vector goes upward or downward

174 vlookup_datf

Examples

```
datf1 <- data.frame(c(1:5), c(5:1), c("a", "z", "z", "z", "a"))</pre>
print(datf1)
# c.1.5. c.5.1. c..a...z...z...z.....z....a..
#1
       1
#2
        2
               4
#3
       3
               3
              2
#4
       4
       5
              1
#5
print(vec_in_datf(inpt_datf=datf1, inpt_vec=c(5, 4, "z"), coeff=1))
#NULL
print(vec_in_datf(inpt_datf=datf1, inpt_vec=c(5, 2, "z"), coeff=1))
#[1] 5 1
print(vec_in_datf(inpt_datf=datf1, inpt_vec=c(3, "z"), coeff=1))
#[1] 3 2
print(vec_in_datf(inpt_datf=datf1, inpt_vec=c(4, "z"), coeff=-1))
#[1] 2 2
print(vec_in_datf(inpt_datf=datf1, inpt_vec=c(2, 3, "z"), coeff=-1))
#[1] 2 1
print(vec_in_datf(inpt_datf=datf1, inpt_vec=c(5, 2, "z"), coeff=-1, conventional=TRUE))
#[1] 5 1
datf1[4, 2] <- 1
print(vec_in_datf(inpt_datf=datf1, inpt_vec=c(1, "z"), coeff=-1, conventional=TRUE, stop_
#[1] 4 2 5 2
```

vlookup_datf

vlookup_datf

Description

Alow to perform a vlookup on a dataframe

Usage

```
vlookup_datf(datf, v_id, col_id = 1, included_col_id = "yes")
```

wider_datf 175

Arguments

```
is the input dataframe
datf
v_id
                   is a vector containing the ids
col id
                   is the column that contains the ids (default is equal to 1)
included col id
                   is if the result should return the col_id (default set to yes)
```

Examples

```
datf1 <- data.frame(c("az1", "az3", "az4", "az2"), c(1:4), c(4:1))</pre>
print(vlookup_datf(datf=datf1, v_id=c("az1", "az2", "az3", "az4")))
    c..az1....az3....az4....az2.. c.1.4. c.4.1.
#2
                               az1
                                       1
#4
                                        4
                                               1
                               az2
#21
                               az3
                                        2
                                               3
#3
                               az4
                                        3
                                               2
```

wider_datf wider_datf

Description

Takes a dataframe as an input and the column to split according to a seprator.

Usage

```
wider_datf(inpt_datf, col_to_splt = c(), sep_ = "-")
```

Arguments

```
inpt_datf
                  is the input dataframe
                  is a vector containing the number or the colnames of the columns to split ac-
col_to_splt
                  cording to a separator
                  is the separator of the elements to split to new columns in the input dataframe
sep_
```

Examples

#k-11 4

```
datf1 \leftarrow data.frame(c(1:5), c("o-y", "hj-yy", "er-y", "k-ll", "ooo-mm"), c(5:1))
{\tt datf2} < - \ {\tt data.frame("col1"=c(1:5), "col2"=c("o-y", "hj-yy", "er-y", "k-ll", "ooo-mm"))}
print(wider_datf(inpt_datf=datf1, col_to_splt=c(2), sep_="-"))
        pre_datf X.o. X.y.
#о-у
        1
               "o" "y" 5
#hj-yy 2
               "hj" "yy" 4
               "er" "y" 3
#er-y 3
               "k" "11" 2
```

176 wide_to_narrow_idx

```
#000-mm 5
              "000" "mm" 1
print(wider_datf(inpt_datf=datf2, col_to_splt=c("col2"), sep_="-"))
       pre_datf X.o. X.y.
              "о" "у"
#о-у
       1
              "hj" "yy"
#hj-yy 2
              "er" "y"
#er-y
      3
#k-11 4
              "k" "11"
              "000" "mm"
#000-mm 5
```

```
wide_to_narrow_idx wide_to_narow_idx
```

Description

Allow to convert the indices of vector ('from_v_ids') which are related to each characters of a vector, to fit the newly established maximum character of the vector, see examples.

Usage

```
wide_to_narrow_idx(from_v_val = c(), from_v_ids = c(), val = 1)
```

Arguments

```
from_v_val is the input vector of elements, or just the total number of characters of the
    elementsq in the vector

from_v_ids is the input vector of indices

val is the value - 1 from which the number of character of an element is too high, so
    the indices in 'from_v_ids' will be modified
```

```
print(wide_to_narrow_idx(from_v_val = c("oui", "no", "oui"), from_v_ids = c(4, 6, 9), val
[1] 2 4 5

print(wide_to_narrow_idx(from_v_val = c("oui", "no", "oui"), from_v_ids = c(4, 6, 9), val
[1] 2 2 3

print(wide_to_narrow_idx(from_v_val = c("oui", "no", "oui"), from_v_ids = c(4, 6, 9), val
[1] 4 6 9
```

write_edm_parser 177

```
write_edm_parser write_edm_parser
```

Description

Allow to write data to edm parsed dataset, see examples

Usage

```
write_edm_parser(inpt, to_write_v, write_data)
```

Arguments

```
inpt is the input dataset
to_write_v is the vector containing the path to write the data, see examples
```

```
print(write_edm_parser("(ok(ee:56)) (ok(oui(rr((rr2:6) (rr:5)))) (oui(bb(rr2:1))) (ee1:4))",
to_write_v = c("ok", "ee"), write_data = c("ii", "olm")))

[1] "(ok(ee:56) (ii:olm)) (ok(oui(rr((rr2:6) (rr:5)))) (oui(bb(rr2:1))) (ee1:4))"

print(write_edm_parser("(ok(ee:56)) (ok(oui(rr((rr2:6) (rr:5)))) (oui(bb(rr2:1))) (ee1:4))",
to_write_v = c("ok", "oui"), write_data = c("ii", "olm")))

[1] "(ok(ee:56)) (ok(oui(rr((rr2:6) (rr:5)))) (ii:olm) (oui(bb(rr2:1))) (ee1:4))"

print(write_edm_parser("(ok(ee:56)) (ok(oui(rr((rr2:6) (rr:5)))) (oui(bb(rr2:1))) (ee1:4))",
to_write_v = c("ok", "oui", "oui"), write_data = c("ii", "olm")))

[1] "(ok(ee:56)) (ok(oui(rr((rr2:6) (rr:5)))) (oui(bb(rr2:1))) (ii:olm) (ee1:4))"

print(write_edm_parser("",
to_write_v = c(), write_data = c("ii", "olm")))

[1] "(ii:olm)"
```

Index

all_concat, 5	delta_normal,42
all_stat,6	delta_unif,43
and_bool1,7	depth_pairs_findr,44
and_bool2, 7	diff_datf,44
any_join_datf,8	dynamic_idx_convertr,45
appndr, 10	
arroundr_mean, 11	edm1_random_val,46
arroundr_min, 11	edm1_random_val_spe,46
	edm1_rnorm1,47
better_match, 12	edm1_rnorm2,49
better_split, 13	edm1_runif_deterministic,51
better_split_any, 13	edm1_unif_time, 52
better_sub, 14	edm_arrangr,53
better_sub_mult, 15	edm_arrangr2,54
better_unique, 16	edm_group_by1,55
bind_cols, 17	edm_group_by2,56
bind_rows, 18	edm_pert, 57
	edm_pivot_longer1,58
can_be_num, 18	edm_pivot_longer2,61
closer_ptrn, 19	edm_pivot_series, 64
closer_ptrn_adv,21	edm_pivot_wider1,66
clusterizer_v, 23	edm_pivot_wider2,67
col_convertr, 26	elements_equalifier,69
col_to_row, 27	equalizer_v,70
colins_datf, 25	extract_normal, 70
converter_date, 27	extrt_only_v, 76
converter_format, 28	1
cost_and_taxes, 29	fillr,77
cumulated_rows, 30	fixer_nest_v,78
cumulated_rows_na,31	fold_rec, 78
cut_v, 32	fold_rec2, 79
cutr_v, 31	format_date, 79
– /	
data_gen, 33	$geo_min, 80$
data_meshup, 35	get_rec, 81
date_addr, 36	globe, 81
date_converter_reverse, 37	glue_groupr_v, 82
datf_appendr, 38	grep_all, 82
datf_appendr2, 38	grep_all2, 83
datf_insertr, 39	groupr_datf, 84
datf_row_appendr, 40	gsub_mult, 85
datf_row_appendr2,40	J
dcr_untl, 41	historic_sequence1,85
dcr_val, 42	historic_sequence2, 86
_ ′	,

INDEX 179

id_keepr, 88	pairs_findr, 120
incr_fillr,89	pairs_findr_merger, 120
infinite_char_seq,90	pairs_insertr, 122
inner_all, 90	pairs_insertr2, 123
insert_datf, 91	paste_datf, 125
inter_max, 95	paste_datf2, 125
inter_min, 96	pattern_generator, 126
intersect_all,92	pattern_gettr, 126
intersect_mod, 92	pattern_tuning, 127
intersect_mod2, 94	power_to_char, 128
is_divisible, 97	pre_to_post_idx, 129
isnt_divisible, 97	ptrn_switchr, 129
15110_411151515,77	ptrn_twkr, 130
join_n_lvl,98	perii_ewkr, 150
just_anything, 99	r_print, 136
just_anything2,99	read_edm_parser, 131
just_anything3, 100	rearangr_v, 132
just_chr, 100	regex_spe_detect, 132
_	regroupr, 133
just_chr2, 101	
just_chr3, 101	rm_na_rows, 134
just_nb, 102	rm_rows, 135
just_nb2, 102	row_to_col, 136
just_nb3, 103	garra un+1 127
just_not_anything, 103	save_untl, 137
just_not_anything2, 104	see_datf, 138
just_not_anything3,104	see_diff, 139
	see_diff_all, 140
leap_yr, 105	see_diff_detailled, 140
left_all, 105	see_file, 141
letter_to_nb, 106	see_idx, 141
list_files, 106	see_in_grep, 143
lst_flatnr, 107	see_in_1, 143
	see_inside, 142
match_by, 107	see_mode, 144
match_na_omit, 108	selected_char, 144
multitud, 108	sequence_na_mean1, 145
mutate_vector, 109	sequence_na_mean2,146
	sequence_na_med1, 147
nb2_follow, 110	sequence_na_med2, 148
nb_follow, 111	sort_date, 149
nb_to_letter, 111	sort_normal_qual, 150
nest_v, 114	sort_normal_qual2,154
nestr_datf1,113	split_by_step, 157
nestr_datf2, 114	str_remove_untl, 158
new_ordered, 115	sub_mult, 159
normal_dens, 116	successive_diff, 160
normal_offset_prob, 116	sum_group1, 160
normal_offset_val, 117	sum_group2, 162
	swipr, 164
occu, 118	÷ /
old_to_new_idx, 118	test_order, 165
or_bool1, 119	time_serie_equalizer, 165
or_bool2, 119	to_unique, 167

INDEX

```
union_all, 168
union_keep, 168
unique_datf, 169
unique_ltr_from_v, 170
unique_pos, 170
unique_total, 171
until_stnl, 171

val_replacer, 172
vec_in_datf, 173
vector_replacor, 172
vlookup_datf, 174

wide_to_narrow_idx, 176
wider_datf, 175
write_edm_parser, 177
```