

Package ‘edm1’

August 9, 2024

Title Simplify Complex Data Manipulation

Version 2.0.0.0

Description Provides complex sorting algorithms. Provides date manipulation algorithms. 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_stat	5
any_join_datf	6
appndr	8
arroundr_mean	9
arroundr_min	9
better_match	10
better_split	11
better_split_any	11
better_sub	12
better_sub_mult	13
better_unique	14
bind_cols	15
bind_rows	16
can_be_num	16
closer_ptrn	17

closer_ptm_adv	19
clusterizer_v	21
colins_datf	23
col_to_row	24
converter_date	25
converter_format	26
cost_and_taxes	26
cumulated_rows	27
cumulated_rows_na	28
cutr_v	29
cut_v	29
data_gen	30
data_meshup	32
date_addr	33
date_converter_reverse	34
datf_appendr	35
datf_appendr2	35
datf_insertr	36
datf_row_appendr	37
datf_row_appendr2	37
dcr_untl	38
dcr_val	39
depth_pairs_findr	39
diff_datf	40
dynamic_idx_converttr	40
edm_group_by1	41
edm_group_by2	42
edm_pivot_longer1	43
edm_pivot_longer2	45
edm_pivot_wider1	47
edm_pivot_wider2	49
elements_equalifier	50
equalizer_v	51
extract_normal	52
extrt_only_v	58
fillr	58
fixer_nest_v	59
fold_rec	60
fold_rec2	60
format_date	61
geo_min	61
get_rec	62
globe	62
glue_groupr_v	63
grep_all	64
grep_all2	64
groupr_datf	65
gsub_mult	66
historic_sequence1	67
historic_sequence2	68
how_normal	69
how_unif	72

id_keepr	73
incr_fillr	74
infinite_char_seq	75
inner_all	75
insert_datf	76
intersect_all	77
intersect_mod	77
inter_max	79
inter_min	80
isnt_divisible	81
is_divisible	81
join_n_lvl	82
just_anything	83
just_anything2	83
just_anything3	84
just_chr	84
just_chr2	85
just_chr3	85
just_nb	86
just_nb2	86
just_nb3	87
just_not_anything	87
just_not_anything2	88
just_not_anything3	88
leap_yr	89
left_all	89
letter_to_nb	90
list_files	90
lst_flatnr	91
match_by	91
multitud	92
nb2_follow	93
nb_follow	93
nb_to_letter	94
nestr_datf1	95
nestr_datf2	96
nest_v	97
new_ordered	98
normal_dens	98
occu	99
old_to_new_idx	99
pairs_findr	100
pairs_findr_merger	100
pairs_insertr	102
pairs_insertr2	103
paste_datf	105
paste_datf2	105
pattern_generator	106
pattern_gettr	106
pattern_tuning	107
power_to_char	108
pre_to_post_idx	109

ptrn_switchr	109
ptrn_twkr	110
read_edm_parser	111
rearangr_v	112
regex_spe_detect	112
regroupr	113
rm_na_rows	114
rm_rows	115
row_to_col	116
r_print	116
save_untl	117
see_datf	118
see_diff	119
see_diff_all	120
see_file	120
see_idx	121
see_inside	121
see_in_grep	122
see_in_l	123
see_mode	123
selected_char	124
sequence_na_mean1	124
sequence_na_mean2	125
sequence_na_med1	126
sequence_na_med2	127
sort_date	129
sort_normal_qual	130
sort_normal_qual2	133
split_by_step	137
str_remove_untl	138
sub_mult	138
successive_diff	139
swipr	140
test_order	140
to_unique	141
union_all	142
union_keep	142
unique_datf	143
unique_ltr_from_v	144
unique_pos	144
unique_total	145
until_stnl	146
val_replacer	146
vector_replacor	147
vec_in_datf	147
vlookup_datf	149
wider_datf	149
wide_to_narrow_idx	150
write_edm_parser	151

all_stat	<i>all_stat</i>
----------	-----------------

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 occurrence 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

Examples

```
datf <- data.frame("mod"=c("first", "seco", "seco", "first", "first", "third", "first"),
  "var1"=c(11, 22, 21, 22, 22, 11, 9),
  "var2"=c("d", "d", "z", "z", "z", "d", "z"),
  "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          var1      64   16 16.5   6.97614984548545 48.66666666666667
#3          var2-d       1
#4          var2-z       3
#5          var3     178 44.5   45   1.73205080756888           3
#6          var4-A       2
#7          var4-B       1
#8          var4-C       1
#9      seco
#10         var1      43 21.5 21.5   0.707106781186548           0.5
#11         var2-d       1
#12         var2-z       1
#13         var3     87 43.5 43.5   0.707106781186548           0.5
#14         var4-A       2
#15         var4-B       0
#16         var4-C       0
#   quantile-0.75
```

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

any_join_datf	<i>any_join_datf</i>
---------------	----------------------

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

<code>inpt_datf_l</code>	is a list containing all the dataframe
<code>join_type</code>	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.
<code>join_spe</code>	can be equal to a vector to do an external joints on all the dataframes. In this case, <code>join_type</code> should not be equal to "inner"
<code>id_v</code>	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 <code>inpt_datf_l</code> . 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 $5 + 3 = 8$
<code>excl_col</code>	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 <code>c()</code>

`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

Examples

```
datf1 <- data.frame("val"=c(1, 1, 2, 4), "ids"=c("e", "a", "z", "a"),
  "last"=c("oui", "oui", "non", "oui"),
  "second_ids"=c(13, 11, 12, 8), "third_col"=c(4:1))

datf2 <- 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"),
  "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          13
#3  z  2  z non          12  7  z FALSE          12  2  z non          12
#4  a  4  a oui           8  4  a FALSE          34  9  a oui          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
#1  e  1  e oui          13 <NA> <NA> <NA>          <NA> <NA> <NA> <NA>
#2  a  1  a oui          11  3  a TRUE          13  1  a oui
#3  z  2  z non          12  7  z FALSE          12  2  z non
#4  a  4  a oui           8  4  a FALSE          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
#1 a13  3  a TRUE          13 <NA> <NA> <NA>          <NA>  1  a oui
#2 z12  7  z FALSE          12  2  z non          12  2  z non
#3 z8  2  z FALSE           8 <NA> <NA> <NA>          <NA> <NA> <NA> <NA>
#4 a34  4  a FALSE          34 <NA> <NA> <NA>          <NA> <NA> <NA> <NA>
```

```
#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>
#8 a11 <NA> <NA> <NA> <NA> 1 a oui 11 9 a oui
#9 z12 <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
#10 a8 <NA> <NA> <NA> <NA> 4 a oui 8 4 a oui
# second_ids
#1 13
#2 12
#3 <NA>
#4 <NA>
#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
#1 e 1 e oui 13 <NA> <NA> <NA> <NA> <NA> <NA> <NA>
#2 a 1 a oui 11 3 a TRUE 13 1 a oui
#3 z 2 z non 12 7 z FALSE 12 2 z non
#4 a 4 a oui 8 4 a FALSE 34 9 a oui
# second_ids
#1 <NA>
#2 13
#3 12
#4 11
```

appndr	<i>appndr</i>
--------	---------------

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"

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	<i>arroundr_mean</i>
---------------	----------------------

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

```
x <- arroundr_mean(inpt_v = c(-11:25), step_val = 5)
print(x)
print(length(x))

[1] -9.0 -9.0 -9.0 -9.0 -9.0 -4.0 -4.0 -4.0 -4.0 -4.0  1.0  1.0  1.0  1.0  1.0
[16]  6.0  6.0  6.0  6.0  6.0 11.0 11.0 11.0 11.0 11.0 16.0 16.0 16.0 16.0 16.0
[31] 21.0 21.0 21.0 21.0 21.0 23.8 23.8
[1] 37
```

arroundr_min	<i>arroundr_min</i>
--------------	---------------------

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)
```

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

better_match	<i>better_match</i>
--------------	---------------------

Description

Allow to get the nth element matched in a vector

Usage

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

Arguments

inpt_v is the input vector
ptrn is the pattern to be matched
until is the maximum number of matched pattern outputed
nvr_here is a value you are sure is not present in inpt_v

Examples

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

better_split	<i>better_split</i>
--------------	---------------------

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	<i>better_split_any</i>
------------------	-------------------------

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

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

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"
```

```
print(better_split_any(inpt = "--o--_/m/m/___opo-/m/-u_i_--", split_v = c("--", "_", "/"
```

```
[1] "--"      "o"      "--"      "_"      "/"      "m"      "/"      "m"      "/"
[10] "_"      "_"      "-opo-"  "/"      "m"      "/"      "-u"      "_"      "i-"
[19] "_"      "--"
```

```
print(better_split_any(inpt = "(ok(ee:56))(ok2(oui)(ee:4))", split_v = c("(", ")", ":")))
```

```
[1] "("      "ok"      "("      "ee"      ":"      "56"      ")"      ")"      "("      "ok2"      "("      "oui"
[13] ")"      "("      "ee"      ":"      "4"      ")"      ")"
```

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, until_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 substitute pattern

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

Examples

```
print(better_sub(inpt_v = c("yes NAME, i will call NAME and NAME",
                             "yes NAME, i will call NAME and NAME"),
               pattern = "NAME",
               replacement = "Kevin",
               until = c(2)))
```

```
[1] "yes Kevin, i will call Kevin and NAME"
[2] "yes Kevin, i will call Kevin and NAME"
```

```
print(better_sub(inpt_v = c("yes NAME, i will call NAME and NAME",
                           "yes NAME, i will call NAME and NAME"),
          pattern = "NAME",
          replacement = "Kevin",
          until = c(2, 3)))

[1] "yes Kevin, i will call Kevin and NAME"
[2] "yes Kevin, i will call Kevin and Kevin"

print(better_sub(inpt_v = c("yes NAME, i will call NAME and NAME",
                           "yes NAME, i will call NAME and NAME"),
          pattern = "NAME",
          replacement = "Kevin",
          until = c("max", 3)))

[1] "yes Kevin, i will call Kevin and Kevin"
[2] "yes Kevin, i will call Kevin and Kevin"
```

better_sub_mult	<i>better_sub_mult</i>
-----------------	------------------------

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(),
  until_v = c()
)
```

Arguments

<code>inpt_v</code>	is a vector containing all the elements that contains expressions to be substituted
<code>pattern_v</code>	is a vector containing all the patterns to be substituted in any elements of <code>inpt_v</code>
<code>replacement_v</code>	is a vector containing the expression that are going to substitute those provided by <code>pattern_v</code>
<code>until_v</code>	is a vector containing, for each element of <code>inpt_v</code> , the number of pattern that will be substituted

Examples

```
print(better_sub_mult(inpt_v = c("yes NAME, i will call NAME and NAME2",
                                "yes NAME, i will call NAME and NAME2, especially NAME2"),
  pattern_v = c("NAME", "NAME2"),
  replacement_v = c("Kevin", "Paul"),
  until = c(1, 3)))
```

```
[1] "yes Kevin, i will call NAME and Paul"
[2] "yes Kevin, i will call NAME and Paul, especially Paul"

print(better_sub_mult(inpt_v = c("yes NAME, i will call NAME and NAME2",
                                "yes NAME, i will call NAME and NAME2, especially NAME2"),
                    pattern_v = c("NAME", "NAME2"),
                    replacement_v = c("Kevin", "Paul"),
                    until = c("max", 3)))

[1] "yes Kevin, i will call Kevin and Kevin2"
[2] "yes Kevin, i will call Kevin and Kevin2, especially Kevin2"
```

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 occurrence 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 syntax is the following "comparaison_type-actual_value-". The comparaison type may be "==" or ">" or "<". Occu can also be a vector containing all the occurrence that must have the elements to be returned.

Examples

```
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("non", "peut", "peut1"))
#[1] "non" "peut" "peut1"
```

```
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"
```

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 contains the cols to find in `from_datf`

Examples

```
iris[, 5] <- as.character(iris[, 5])
iris <- cbind(iris, iris[, 4])
from_datf <- iris
in_datf <- iris[, c(1, 2, 2, 2, 4)]
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
```

bind_rows	<i>bind_rows</i>
-----------	------------------

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 contains the rows to find in from_datf

Examples

```
iris[, 5] <- as.character(iris[, 5])
from_datf <- iris
in_datf <- iris[c(4, 2, 23, 21, 11), ]

bind_rows(from_datf = from_datf,
          in_datf = in_datf)

[[1]]
[1] 4

[[2]]
[1] 2

[[3]]
[1] 23

[[4]]
[1] 21

[[5]]
[1] 11
```

can_be_num	<i>can_be_num</i>
------------	-------------------

Description

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

Usage

```
can_be_num(x)
```


Arguments

`x` 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

<code>inpt_v</code>	is the input vector containing all the patterns
<code>base_v</code>	must contain all the characters that the patterns are susceptible to contain, defaults to <code>c("?", letters)</code> . "?" 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 <code>inpt_v</code> . If set to NA, the function will find by itself the elements to be filled with but it may take an extra time
<code>excl_v</code>	is the vector containing all the patterns from <code>inpt_v</code> to exclude for comparing them to others patterns. If this parameter is filled, so <code>"rtn_v"</code> must be empty.

`rtn_v` 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

```
print(closer_ptrn(inpt_v=c("bonjour", "lpoerc", "nonnour", "bonnour", "nonjour", "aurevoir"),
  # [[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"
  #
  # [[15]]
```

```

#[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", "aurevoir",
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

```

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

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

Examples

```
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	<i>clusterizer_v</i>
---------------	----------------------

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 number
w_v	is the vector containing the elements related to inpt_v, defaults to NA
c_val	is the accuracy of the clusterization

Examples

```
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
```

```

#
#[[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] "l"
#
#[[1]][[8]]
#[1] "m" "n"
#
#[[1]][[9]]
#[1] "o"
#
#[[1]][[10]]
#[1] "p"
#
#[[1]][[11]]
#[1] "q" "r"

```

```

#
#[[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

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

Examples

```

datf1 <- data.frame("frst_col"=c(1:5), "scd_col"=c(5:1))

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("frst_col", "scd_col"), c("scd_col"))))

```

```
# first_col cur_col scd_col cur_col.1 cur_col
#1      1      oui      5      oui      u
#2      2      oui      4      oui      z
#3      3      oui      3      oui      z
#4      4      non      2      non      z
#5      5      non      1      non      u

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("first_col"))))

# first_col cur_col scd_col cur_col cur_col
#1      1      oui      5      u      oui
#2      2      oui      4      z      oui
#3      3      oui      3      z      oui
#4      4      non      2      z      non
#5      5      non      1      u      non
```

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
```

converter_date	<i>converter_date</i>
----------------	-----------------------

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_ = "-")
```

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	<i>converter_format</i>
------------------	-------------------------

Description

Allow to convert a format to another

Usage

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

Arguments

inpt_val	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

Examples

```
print(converter_format(inpt_val="23-12-05-1567", sep_="-",
                        inpt_frmt="shmy", frmt="snhdmy", default_val="00"))

#[1] "23-00-12-00-05-1567"

print(converter_format(inpt_val="23-12-05-1567", sep_="-",
                        inpt_frmt="shmy", frmt="Pnhdmy", default_val="00"))

#[1] "00-00-12-00-05-1567"
```

cost_and_taxes	<i>cost_and_taxes</i>
----------------	-----------------------

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_ttc = NA
  )

```

Arguments

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

Examples

```

print(cost_and_taxes(pu=45, prix_ttc=2111, qte=23))

# [1] 23.000000 45.000000 45.000000 1.039614 2111.000000 1076.000000
# [7] 45.000000 NA NA NA NA NA

```

cumulated_rows	<i>cumulated_rows</i>
----------------	-----------------------

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

Examples

```
datf_teste <- 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       7
5         5       6
6         6       5
7         7       4
8         8       3
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  FALSE TRUE TRUE  FALSE
```

cumulated_rows_na	<i>cumulated_rows_na</i>
-------------------	--------------------------

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                         10
2                         2                         9
3                         3                         8
4                         4                         NA
5                         5                         7
6                         NA                        6
7                         7                         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, until = "min")
```

Arguments

inpt_v is the input vector

until is the maximum size of nchar authorized by an element, defaults to "min", it means the shortest element in the list

Examples

```
test_v <- c("oui", "nonon", "ez", "aa", "a", "dsfsdsds")

print(cutr_v(inpt_v=test_v, until="min"))

#[1] "o" "n" "e" "a" "a" "d"

print(cutr_v(inpt_v=test_v, until=3))

#[1] "oui" "non" "ez" "aa" "a" "dsf"
```

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 ""

Examples

```
print(cut_v(inpt_v=c("oui", "non", "oui", "non")))

#      X.o. X.u. X.i.
#oui  "o"  "u"  "i"
#non  "n"  "o"  "n"
#oui  "o"  "u"  "i"
#non  "n"  "o"  "n"

print(cut_v(inpt_v=c("ou-i", "n-on", "ou-i", "n-on"), sep_="-"))

#      X.ou. X.i.
#ou-i  "ou"  "i"
#n-on  "n"   "on"
#ou-i  "ou"  "i"
#n-on  "n"   "on"
```

data_gen

data_gen

Description

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

Usage

```
data_gen(
  type_ = c("number", "mixed", "string"),
  strt_l = c(0, 0, 10),
  nb_r = c(50, 10, 40),
  output = NA,
  properties = c("1-5", "1-5", "1-5"),
  type_distri = c("random", "random", "random"),
  str_source = c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m",
    "o", "p", "q", "r", "s", "t", "u", "w", "x", "y", "z"),
  round_l = c(0, 0, 0),
  sep_ = ",",
)
```

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 outputted 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 means that not 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

Examples

```
print(data_gen())
```

```
#   X1   X2   X3
#1   4    2 <NA>
#2   2    4 <NA>
#3   5    2 <NA>
#4   2 abcd <NA>
#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>    a
#17  1 <NA> abcd
#18  4 <NA>   ab
#19  2 <NA> abcd
#20  3 <NA>   ab
#21  3 <NA> abcd
#22  2 <NA>    a
#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>    a
```

```
#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>    ab
#42 3 <NA>    ab
#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>    a
```

```
print(data_gen(strt_l=c(0, 0, 0), nb_r=c(5, 5, 5)))
```

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

data_mesup

data_mesup

Description

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

Usage

```
data_mesup(
  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: <code>c("", c("d", "-", "e", "-", "f"), "", c("a", "a1", "-", "b", "-", "c", "c1"), "-")</code>)
cols	are the colnames of the data generated in a csv
file_	is the file to which the data will be outputed, defaults to NA which means that the funcio will return the dataframe generated and won't write it to a csv file

sep_ is the separator of the csv outputed

organisation is the way variables include themselves, for instance ,resuming precedent example, if organisation=c(1, 0) so the data output will be: d, a d, a1 e, c f, c f, c1

unic_sep1 is the unic separator between variables (default is "_")

unic_sep2 is the unic separator between datasets (default is "-")

Examples

```
print(data_meshup(data=c("_", c("-", "d", "-", "e", "-", "f"), "_",
  c("-", "a", "a1", "-", "B", "r", "uy", "-", "c", "c1"), "_"), organisation=c(1, 0)))

#  X1 X2
#1  d  a
#2  d a1
#3  e  B
#4  e  r
#5  e uy
#6  f  c
#7  f c1
```

date_addr

date_addr

Description

Allow to add or subtract 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 subtracted

date2 is the date that will be added or will subtract 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

Examples

```

print(date_addr(date1="25-02", date2="58-12-08", frmt1="dm", frmt2="shd", sep_="-",
               convert_to="dmy"))

#[1] "18-2-0"

print(date_addr(date1="25-02", date2="58-12-08", frmt1="dm", frmt2="shd", sep_="-",
               convert_to="dmy", add=TRUE))

#[1] "3-3-0"

print(date_addr(date1="25-02-2024", date2="1-01", frmt1="dmy", frmt2="dm", sep_="-",
               convert_to="dmy", add=TRUE))

#[1] "27-3-2024"

print(date_addr(date1="25-02-2024", date2="1-01", frmt1="dmy", frmt2="dm", sep_="-",
               convert_to="dmy", add=FALSE))

#[1] "23-1-2024"

print(date_addr(date1="25-02-2024", date2="1-01", frmt1="dmy", frmt2="dm", sep_="-",
               convert_to="n", add=FALSE))

#[1] "1064596320"

print(date_addr(date1="25-02-2024", date2="1-01", frmt1="dmy", frmt2="dm", sep_="-",
               convert_to="s", add=FALSE))

#[1] "63875779200"

```

```

date_converter_reverse
      date_converter_reverse

```

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

Examples

```
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="-"))

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

datf_appendr	<i>datf_appendr</i>
--------------	---------------------

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
```

datf_appendr2	<i>datf_appendr2</i>
---------------	----------------------

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

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"
```

<code>datf_insertr</code>	<i><code>datf_insertr</code></i>
---------------------------	----------------------------------

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 every index within `ids_vec`

Examples

```
datf <- 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      4      1

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

  c.1.4. c.4.1.
1      1      4
2      2      3
3      3      2
4      4      1
5     non     non
6     non     non
```

```
1      1      4
2      non    non
21     2      3
3      3      2
5      oui    oui
4      4      1

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

c.1.4. c.4.1.
1      1      4
2      non    non
21     2      3
3      3      2
5      non    non
4      4      1
```

<code>datf_row_appendr</code>	<i><code>datf_row_appendr</code></i>
-------------------------------	--------------------------------------

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_teste <- data.frame("col1" = c(1:5), "col2" = c(5:1))

print(datf_appendr(inpt_datf = datf_teste))

col1 col2 col1 col2 col1 col2 col1 col2 col1 col2
1     5     2     4     3     3     4     2     5     1
```

<code>datf_row_appendr2</code>	<i><code>datf_row_appendr2</code></i>
--------------------------------	---------------------------------------

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

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"
```

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

Examples

```
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
```

dcr_val	<i>dcr_val</i>
---------	----------------

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
```

depth_pairs_findr	<i>depth_pairs_findr</i>
-------------------	--------------------------

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	<i>diff_datf</i>
-----------	------------------

Description

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

Usage

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

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"), 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
```

dynamic_idx_converttr	<i>dynamic_idx_converttr</i>
-----------------------	------------------------------

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_converttr(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

Examples

```
print(dynamic_idx_convertertr(from_v_ids = c(1, 5), from_v_val = c("oui", "no", "ouI")))

[1] 1 2

print(dynamic_idx_convertertr(from_v_ids = c(1, 6), from_v_val = c("oui", "no", "ouI")))

[1] 1 3
```

<code>edm_group_by1</code>	<i><code>edm_group_by1</code></i>
----------------------------	-----------------------------------

Description

Performs a group by (different algorithm 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

Examples

```
datf <- 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
1    A    E    P
2    B    R    P
3    B    E    O
4    A    E    O
5    C    R    P
6    B    R    O

print(edm_group_by1(inpt_datf = datf, grp_v = c("col1")))
```

```
  col1 col2 col3
1    A    E    P
```

```

4      A      E      O
2      B      R      P
3      B      E      O
6      B      R      O
5      C      R      P

```

```
print(edm_group_by1(inpt_datf = datf, grp_v = c("col1", "col2")))
```

```

col1 col2 col3
1      A      E      P
4      A      E      O
2      B      R      P
6      B      R      O
3      B      E      O
5      C      R      P

```

```
print(edm_group_by1(inpt_datf = datf, grp_v = c("col2", "col1", "col3")))
```

```

col2 col1 col3
1      E      A      P
4      E      A      O
3      E      B      O
2      R      B      P
6      R      B      O
5      R      C      P

```

```
print(edm_group_by1(inpt_datf = datf, grp_v = c("col2", "col1", "col3")))
```

```

col2 col1 col3
1      E      A      P
4      E      A      O
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 algorithm that edm_group_by1), see examples

Usage

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

Arguments

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

Examples

```

datf <- 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
1    A    E    P
2    B    R    P
3    B    E    O
4    A    E    O
5    C    R    P
6    B    R    O

print(edm_group_by2(inpt_datf = datf, grp_v = c("col1")))

  col1 col2 col3
1    A    E    P
4    A    E    O
2    B    R    P
3    B    E    O
6    B    R    O
5    C    R    P

print(edm_group_by2(inpt_datf = datf, grp_v = c("col1", "col2")))

  col1 col2 col3
1    A    E    P
4    A    E    O
3    B    E    O
2    B    R    P
6    B    R    O
5    C    R    P

print(edm_group_by2(inpt_datf = datf, grp_v = c("col2", "col1")))

  col2 col1 col3
1    E    A    P
4    E    A    O
3    E    B    O
2    R    B    P
6    R    B    O
5    R    C    P

```

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...

Usage

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

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

Examples

```
datf <- 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",
                                  "val1-A.T",
                                  "val1-B.R",
                                  "val1-B.T",
                                  "val2-A.R",
                                  "val2-A.T",
                                  "val2-B.R",
                                  "val2-B.T")

datf2 <- 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(
  "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         1         6         7         0         1         3         9
2             2         2         0         0         0         0         0         0
3             3         3         2         0         0         4         8         0
      val2-B.T
1             11
2              0
3              5

```

```

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             1     A   R     1     1
2             1     A   T     6     3
3             1     B   R     7     9
4             1     B   T     0    11
5             2     A   R     2     0
6             3     A   R     3     4
7             3     A   T     2     8
8             3     B   T     0     5

```

```
print(datf2)
```

```

      individuals val1-A val1-B val2-A val2-B
1             1         7         1         9        11
2             2         0         0         0        22
3             3         2         4         8         5

```

```

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             1     A     7     9
2             1     B     1    11
3             2     B     0    22
4             3     A     2     8
5             3     B     4     5

```

```
edm_pivot_longer2  edm_pivot_longer2
```

Description

Performs a pivot longer on dataframe, 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(), individual_col)
```

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

Examples

```
datf <- 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",
                                "val1-A.T",
                                "val1-B.R",
                                "val1-B.T",
                                "val2-A.R",
                                "val2-A.T",
                                "val2-B.R",
                                "val2-B.T")

datf2 <- 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(
  "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	1	6	7	0	1	3	9
2	2	2	0	0	0	0	0	0
3	3	3	2	0	0	4	8	0
	val2-B.T							
1	11							
2	0							

```

3          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 vall val2
1           1     A   R    1    1
2           1     A   T    6    3
3           1     B   R    7    9
4           1     B   T    0   11
5           2     A   R    2    0
6           2     A   T    0    0
7           2     B   R    0    0
8           2     B   T    0    0
9           3     A   R    3    4
10          3     A   T    2    8
11          3     B   R    0    0
12          3     B   T    0    5

print(datf2)

  individuals vall-A vall-B val2-A val2-B
1           1      7      1      9     11
2           2      0      0      0     22
3           3      2      4      8      5

print(edm_pivot_longer2(inpt_datf = datf2,
                        col_vars = c(2:5),
                        individual_col = 1,
                        col_vars_to = c("Shape")))

  individuals Shape vall val2
1           1     A    7    9
2           1     B    1   11
3           2     A    0    0
4           2     B    0   22
5           3     A    2    8
6           3     B    4    5

```

```
edm_pivot_wider1    edm_pivot_wider1
```

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 containig 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

Examples

```
datf2 <- 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 <- 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	1	A	R	6	3
2	1	A	T	7	9
3	1	B	T	1	11
4	2	B	R	0	22
5	3	B	T	4	5
6	3	A	R	2	8

```
print(datf2)
```

	individual	var1	val1	val2
1	1	A	6	3
2	1	A	7	9
3	1	B	1	11
4	2	B	0	22
5	3	B	4	5
6	3	A	2	8

```
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	1	6	7	0	1	3	9	0
2	2	0	0	0	0	0	0	22
3	3	2	0	0	4	8	0	0
val2-B.T								
1	11							
2	0							
3	5							


```
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	1	7	1	9	11
2	2	0	0	0	22
3	3	2	4	8	5

edm_pivot_wider2 *edm_pivot_wider2*

Description

Performs a pivot wider to a dataframe with a different algorithm 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 containig 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

Examples

```
datf2 <- 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 <- 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	1	A	R	6	3
2	1	A	T	7	9
3	1	B	T	1	11

```

4          2    B    R    0    22
5          3    B    T    4    5
6          3    A    R    2    8

print(datf2)

  individual var1 vall val2
1          1    A    6    3
2          1    A    7    9
3          1    B    1   11
4          2    B    0   22
5          3    B    4    5
6          3    A    2    8

print(edm_pivot_wider2(
  inpt_datf = datf,
  col_vars = c(2, 3),
  col_vals = c(4, 5),
  individual_col = 1)
)

  individuals vall-A.R vall-A.T vall-B.R vall-B.T val2-A.R val2-A.T val2-B.R
1          1          6          7          0          1          3          9          0
2          2          0          0          0          0          0          22
3          3          2          0          0          4          8          0
  val2-B.T
1         11
2          0
3          5

print(edm_pivot_wider2(
  inpt_datf = datf2,
  col_vars = c(2),
  col_vals = c(3, 4),
  individual_col = 1)
)

  individuals vall-A vall-B val2-A val2-B
1          1          7          1          9          11
2          2          0          0          0          22
3          3          2          4          8          5

```

```
elements_equalifier
```

```
elements_equalifier
```

Description

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

Usage

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

Arguments

`inpt_v` is the input vector
`until` is how many times each elements will be in the output vector

Examples

```
print(elements_equalifier(letters, until = 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]), until = 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"
```

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	<i>extract_normal</i>
----------------	-----------------------

Description

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

Usage

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

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
mean	is the mean of the target normal distribution
sd	is the standard deviation of the target normal distribution
accuracy	is how much of a difference between the points of the targeted normal distribution and the actual points is tolerated
round_value	is the round value for the normal distribution used under the hood to compare the dataset and extract the best points, defaults to 1
normalised	is if the input frequency is divided by n, if TRUE the parameter n must be filled
n	is the number of points
tries	is how many normal distributions are used under the hood to compare their 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.

Examples

```
sample_val <- round(rnorm(n = 72000, 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)
n <- nrow(datf_test)
print(datf_test)

sample_qual sample_freq
```

1	a	72
2	b	1155
3	c	1255
4	d	743
5	e	696
6	f	1028
7	g	1160
8	h	1219
9	i	1353
10	j	1336
11	k	1308
12	l	485
13	m	1306
14	n	1429
15	o	623
16	p	1172
17	q	1054
18	r	999
19	s	125
20	t	1461
21	u	1430
22	v	341
23	w	1453
24	x	427
25	y	869
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	ah	571
35	ai	349
36	aj	773
37	ak	1086
38	al	1281
39	am	1471
40	an	1236
41	ao	394
42	ap	1433
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	bp	10
69	bq	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
83	ce	503
84	cf	164
85	cg	32
86	ch	259
87	ci	1089
88	cj	249
89	ck	165
90	cl	42
91	cm	143
92	cn	467
93	co	347
94	cp	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	cy	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
122	dr	8
123	ds	10
124	dt	4
125	du	53
126	dv	29
127	dw	1
128	dx	8
129	dy	10
130	dz	4
131	ea	22
132	eb	9
133	ec	17
134	ed	55
135	ee	21
136	ef	6
137	eg	4
138	eh	3
139	ei	7
140	ej	1
141	ek	4
142	el	2
143	em	5
144	en	4
145	eo	1
146	ep	2
147	eq	3
148	er	8
149	es	4
150	et	3
151	eu	3
152	ev	2
153	ew	2
154	ex	2
155	ey	1
156	ez	2
157	fa	2
158	fb	1

```

teste <- extract_normal(inpt_datf = datf_test,
                        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
1	dw	0.0001406866
2	dw	0.0001406866
3	dw	0.0001406866
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
13	ay	0.0008441193
14	ay	0.0008441193
15	ei	0.0009848059
16	ei	0.0009848059
17	ei	0.0009848059
18	dm	0.0011254924
19	bp	0.0014068655
20	cy	0.0022509848
21	cy	0.0022509848
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
29	dp	0.0036578503
30	cg	0.0045019696
31	cg	0.0045019696
32	df	0.0047833427
33	dn	0.0050647158
34	cl	0.0059088351
35	cl	0.0059088351
36	du	0.0074563872
37	du	0.0074563872
38	dg	0.0078784468
39	dg	0.0078784468
40	bg	0.0087225661
41	bg	0.0087225661
42	dd	0.0092853123
43	cq	0.0097073720
44	cq	0.0097073720
45	a	0.0101294316
46	cv	0.0120990433
47	cx	0.0123804164
48	cx	0.0123804164
49	bz	0.0136465954
50	cc	0.0140686550
51	bh	0.0150534609
52	bh	0.0150534609
53	dj	0.0168823860

54	s	0.0175858188
55	s	0.0175858188
56	cm	0.0201181767
57	cf	0.0230725943
58	ck	0.0232132808
59	bt	0.0250422060
60	bt	0.0250422060
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
68	ck	0.0232132808
69	ck	0.0232132808
70	cm	0.0201181767
71	cu	0.0198368036
72	s	0.0175858188
73	dj	0.0168823860
74	bh	0.0150534609
75	bh	0.0150534609
76	de	0.0147720878
77	bz	0.0136465954
78	bz	0.0136465954
79	cx	0.0123804164
80	cv	0.0120990433
81	db	0.0105514913
82	a	0.0101294316
83	cq	0.0097073720
84	dd	0.0092853123
85	dd	0.0092853123
86	bg	0.0087225661
87	bg	0.0087225661
88	dg	0.0078784468
89	dk	0.0075970737
90	du	0.0074563872
91	cl	0.0059088351
92	cl	0.0059088351
93	dn	0.0050647158
94	df	0.0047833427
95	df	0.0047833427
96	cg	0.0045019696
97	dv	0.0040799100
98	dp	0.0036578503
99	di	0.0030951041
100	di	0.0030951041
101	ee	0.0029544176
102	cr	0.0025323579
103	dh	0.0023916714
104	cy	0.0022509848
105	cy	0.0022509848
106	cy	0.0022509848
107	cy	0.0022509848
108	dl	0.0012661790
109	dm	0.0011254924
110	ei	0.0009848059

```

111      ei 0.0009848059
112      ay 0.0008441193
113      ay 0.0008441193
114      em 0.0007034328
115      em 0.0007034328
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	<i>extrt_only_v</i>
--------------	---------------------

Description

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

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	<i>fillr</i>
-------	--------------

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.

Examples

```
print(fillr(c("a", "b", "...3", "c")))

#[1] "a" "b" "b" "b" "b" "c"
```

<code>fixer_nest_v</code>	<i><code>fixer_nest_v</code></i>
---------------------------	----------------------------------

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 patterof elements.

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

```
print(fixer_nest_v(cur_v=c("oui", "non", "peut-etre", "oui", "non", "peut-etre"),
  pttrn_v=c("oui", "non", "peut-etre"),
  wrk_v=c(1, 2, 3, 4, 5, 6)))

#[1] 1 2 3 4 5 6

print(fixer_nest_v(cur_v=c("oui", "non", "peut-etre", "oui", "non", "peut-etre"),
  pttrn_v=c("oui", "non"),
  wrk_v=c(1, 2, 3, 4, 5, 6)))

#[1] 1 2 NA 4 5 NA
```

`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 algorithm (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

<code>xmax</code>	is the end depth value
<code>xmin</code>	is the start depth value
<code>pathc</code>	is the reference path

`fold_rec2`*fold_rec2*

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 algorithm that does not need 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

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

format_date	<i>format_date</i>
-------------	--------------------

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 "-")

Examples

```
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	<i>geo_min</i>
---------	----------------

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 coordiantes of the established geographical points

Examples

```
in_ <- data.frame(c(11, 33, 55), c(113, -143, 167))

in2_ <- data.frame(c(12, 55), c(115, 165))

print(geo_min(inpt_datf=in_, established_datf=in2_))

#           X1           X2
#1    245.266         NA
#2 24200.143         NA
#3           NA 127.7004

in_ <- 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_))

#           X1           X2
#1           NA 4343.720
#2 26465.63         NA
#3           NA 5825.517
```

get_rec	<i>get_rec</i>
---------	----------------

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
-------	--

<i>globe</i>	<i>globe</i>
--------------	--------------

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

Examples

```
print(globe(lat_f=23, long_f=112, alt_f=NA, lat_n=c(2, 82), long_n=c(165, -55), alt_n=NA))

#[1] 6342.844 7059.080

print(globe(lat_f=23, long_f=112, alt_f=8, lat_n=c(2, 82), long_n=c(165, -55), alt_n=c(8, 10)),

#[1] 6342.844 7059.087
```

glue_groupv	<i>glue_groupv</i>
-------------	--------------------

Description

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

Usage

```
glue_groupv(inpt_v, group_v = c(), until)
```

Arguments

inpt_v	is the input vector
is	a vector containing all the elements that will be glued in the output vector

Examples

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

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

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

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

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

Examples

```
print(grep_all(inpt_v = c(1:14, "z", 1:7, "z", "a", "z"),
               pattern_v = c("z", "4")))

[1] 15 23 25 4 14 19

print(grep_all(inpt_v = c(1:14, "z", 1:7, "z", "a", "z"),
               pattern_v = c("z", "^4$")))

[1] 15 23 25 4 19

print(grep_all(inpt_v = c(1:14, "z", 1:7, "z", "a", "z"),
               pattern_v = c("z")))

[1] 15 23 25
```

 grep_all2

grep_all2

Description

Performs the grep_all function with another algorithm, 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

Examples

```
print(grep_all2(inpt_v = c(1:14, "z", 1:7, "z", "a", "z"),
               pattern_v = c("z", "4")))

[1] 15 23 25  4 14 19

print(grep_all2(inpt_v = c(1:14, "z", 1:7, "z", "a", "z"),
               pattern_v = c("z", "^4$")))

[1] 15 23 25  4 19

print(grep_all2(inpt_v = c(1:14, "z", 1:7, "z", "a", "z"),
               pattern_v = c("z")))

[1] 15 23 25
```

groupr_datf	<i>groupr_datf</i>
-------------	--------------------

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 dataframe according 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

<code>inpt_datf</code>	is the input dataframe
<code>condition_lst</code>	is a list containing all the condition as a vector for each group
<code>val_lst</code>	is a list containing all the values associated with <code>condition_lst</code> as a vector for each group
<code>conjunction_lst</code>	is a list containing all the conjunctions associated with <code>condition_lst</code> and <code>val_lst</code> as a vector for each group
<code>rtn_val_pos</code>	is a vector containing all the group flag value like this ex: <code>c("flag1", "flag2", "flag3")</code>

Examples

```

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("%%", "==" , "=="))

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    ++  ++  +++
#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

`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 substitute those provided by `pattern_v`

Examples

```

print(gsub_mult(inpt_v = c("X and Y programming languages are great", "More X, more X!"),
  pattern_v = c("X", "Y", "Z"),
  replacement_v = c("C", "R", "GO")))
[1] "C and R programming languages are great"
[2] "More C, more C!"

```

historic_sequence1 *historic_sequence1*

Description

Allow to perform a pivot wider on a sequential dataset (here the type is dataframe), each variable will be duplicated 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

Examples

```
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)

print(datf)
```

	ids	individual	var1	var2
1	20	oui	106	16
2	20	non	117	19
3	20	peut1	109	16
4	20	peut2	119	19
5	19	oui	121	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	peut1	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
1	20	oui	121	120	20	19
2	20	non	NA	112	NA	17

```

3      20      peut1      101      110      14      17
4      20      peut2      112      121      17      20
5      19           oui      120      110      19      17
6      19      peut1      110      115      17      18
7      19      peut2      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      20         non      NA     112      NA      NA      17      NA
3      20      peut1     101     110     115      14      17      18
4      20      peut2     112     121     113      17      20      17

```

```
historic_sequence2 historic_sequence2
```

Description

Allow to perform a pivot wider on a sequential dataset (here the type is dataframe), each variable will be duplicated 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

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

Examples

```

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)

print(datf)

```

```

  ids individual var1 var2
1  20         oui  106   16
2  20         non  117   19
3  20      peut1  109   16
4  20      peut2  119   19
5  19         oui  121   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      peut1  115  18
14 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      20        oui   106   121   120    16    20    19
2      20        non   117    NA   112    19    NA    17
3      20      peut1   109   101   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   110   115    14    17    18
7      19      peut2   112   121   113    17    20    17

```

```
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
1      20        oui   106   121   120   110    16    20    19    17
2      20        non   117    NA   112    NA    19    NA    17    NA
3      20      peut1   109   101   110   115    16    14    17    18
4      20      peut2   119   112   121   113    19    17    20    17

```

how_normal

how_normal

Description

Allow to get how much a sequence of numbers fit a normal distribution with chosen parameters, see examples

Usage

```
how_normal(inpt_datf, normalised = TRUE, mean = 0, sd = 1)
```

Arguments

inpt_datf	is the input dataframe containing all the values in the first column and their frequency (normalised or no), in the second column
normalised	is a boolean, takes TRUE if the frequency for each value is divided by n, FALSE if not
mean	is the mean of the normal distribution that the dataset tries to fit
sd	is the standard deviation of the normal distribution the dataset tries to fit

Examples

```
sample_val <- round(rnorm(n = 12000, mean = 6, sd = 1.25), 1)
sample_freq <- unique_total(sample_val)
datf_test <- data.frame(unique(sample_val), sample_freq)
print(datf_test)
```

	unique.sample_val.	sample_freq
1	6.9	306
2	8.3	63
3	7.7	148
4	5.6	363
5	6.5	349
6	4.6	202
7	6.6	324
8	6.7	335
9	6.0	406
10	5.7	365
11	7.9	109
12	6.2	420
13	5.9	386
14	4.5	185
15	5.1	326
16	6.1	360
17	5.5	346
18	6.3	375
19	7.4	207
20	7.6	162
21	4.2	129
22	3.9	102
23	5.2	325
24	2.3	7
25	5.8	387
26	6.4	319
27	9.1	21
28	7.0	280
29	8.8	27
30	4.9	218
31	8.1	98
32	3.0	25
33	8.4	66
34	4.3	160
35	7.2	267
36	8.7	40
37	5.3	313
38	4.1	127
39	5.0	275
40	4.0	119
41	9.3	13
42	4.4	196
43	6.8	313
44	7.1	247
45	3.5	57
46	7.8	139
47	3.6	57
48	7.5	189
49	7.3	215

50	4.7	230
51	3.2	36
52	9.5	8
53	3.8	79
54	8.2	62
55	5.4	343
56	8.5	55
57	4.8	207
58	3.7	79
59	8.6	33
60	3.3	38
61	3.4	43
62	8.9	21
63	8.0	105
64	3.1	23
65	9.0	27
66	10.0	5
67	2.5	10
68	2.9	16
69	9.7	7
70	2.7	11
71	10.5	1
72	9.4	13
73	9.2	16
74	2.6	16
75	9.9	3
76	2.8	10
77	2.4	10
78	1.9	2
79	2.0	6
80	10.2	2
81	9.6	3
82	11.3	1
83	1.8	1
84	2.2	3
85	2.1	2
86	1.6	1
87	10.6	1
88	9.8	1
89	10.4	1
90	1.7	1

```
print(how_normal(inpt_datf = datf_test,
                 normalised = FALSE,
                 mean = 6,
                 sd = 1))
```

```
[1] 9.003683
```

```
print(how_normal(inpt_datf = datf_test,
                 normalised = FALSE,
                 mean = 5,
                 sd = 1))
```

```
[1] 9.098484
```

how_unif

how_unif

Description

Allow to see how much a sequence of numbers fit a uniform distribution, see examples

Usage

```
how_unif(inpt_v, normalised = TRUE)
```

Arguments

normalised is a boolean, takes TRUE if the frequency for each value is divided by n, FALSE if not

inpt_datf is the input dataframe containing all the values in the first column and their frequencyu at the second column

Examples

```
sample_val <- round(runif(n = 12000, min = 24, max = 27), 1)
sample_freq <- unique_total(sample_val)
datf_test <- data.frame(unique(sample_val), sample_freq)
```

```
print(datf_test)
```

```
  unique.sample_val. sample_freq
1             24.4           400
2             24.8           379
3             25.5           414
4             26.0           366
5             26.6           400
6             25.7           419
7             24.3           389
8             24.1           423
9             26.1           404
10            26.5           406
11            26.2           356
12            26.8           407
13            24.6           388
14            25.3           402
15            26.3           388
16            25.4           422
17            25.0           436
18            25.9           373
19            25.2           423
20            25.6           388
21            27.0           202
22            24.2           380
23            24.9           404
24            25.1           417
25            26.4           401
26            26.7           431
27            24.5           392
```



```

28          24.0          218
29          26.9          407
30          25.8          371
31          24.7          394

print(how_unif(inpt_datf = datf_test, normalised = FALSE))

[1] 0.0752957

sample_val <- round(rnorm(n = 12000, mean = 24, sd = 7), 1)
sample_freq <- unique_total(sample_val)
datf_test <- data.frame(unique(sample_val), sample_freq)

print(how_unif(inpt_datf = datf_test, normalised = FALSE))

[1] 0.7797352

```

id_keepr

id_keepr

Description

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

Usage

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

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 lenght 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

Examples

```

datf1 <- data.frame(c("oui", "oui", "oui", "non", "oui"),
  c("opui", "op", "op", "zez", "zez"), c(5:1), c(1:5))

print(id_keepr(inpt_datf=datf1, col_v=c(1, 2), el_v=c("oui", "op")))

#[1] 2 3

print(id_keepr(inpt_datf=datf1, col_v=c(1, 2), el_v=c("oui", "op"),

```

```

      rstr_l=list(c(1:5), c(3, 2, 2, 2, 3)))

#[1] 2 3

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

#[1] 3

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

Examples

```

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"))

```

```
#[1] "1" "2" "NAN" "4" "5" "NAN" "NAN" "NAN" "9" "10"
```

```
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

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

```

insert_datf	<i>insert_datf</i>
-------------	--------------------

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 beginning for the insertion

Examples

```

datf1 <- data.frame(c(1, 4), c(5, 3))

datf2 <- 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                  5
# 2              3      2                  4
# 3              5      3                  5
# 4              6      1                  5

print(insert_datf(datf_in=datf2, datf_ins=datf1, ins_loc=c(3, 2)))

```

```
#   c.1..3..5..6. c.1.4. c.5..4..5...ereer..
# 1             1     1                      5
# 2             3     2                      4
# 3             5     1                      5
# 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             1     1                      5
# 2             3     1                      5
# 3             5     4                      3
# 4             6     4                      ereer
```

intersect_all	<i>intersect_all</i>
---------------	----------------------

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	<i>intersect_mod</i>
---------------	----------------------

Description

Returns the mods that have elements in common

Usage

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

Arguments

`datf` is the input dataframe

`inter_col` is the column name or the column number of the values that may be commun between the different mods

`mod_col` is the column name or the column number of the mods in the dataframe

`n_min` is the minimum elements in common a mod should have to be taken in count

`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

```
datf <- data.frame("col1"=c("oui", "oui", "oui", "oui", "oui", "oui",
                             "non", "non", "non", "non", "ee", "ee", "ee"), "col2"=c(1:6, 2:5, 1:6))

print(intersect_mod(datf=datf, inter_col=2, mod_col=1, n_min=2))

  col1 col2
2   oui   2
3   oui   3
7   non   2
8   non   3
12  ee    2
13  ee    3

print(intersect_mod(datf=datf, inter_col=2, mod_col=1, n_min=3))

  col1 col2
2   oui   2
3   oui   3
4   oui   4
5   oui   5
7   non   2
8   non   3
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
3   oui   3
4   oui   4
5   oui   5
6   oui   6

datf <- data.frame("col1"=c("non", "non", "oui", "oui", "oui", "oui",
                             "non", "non", "non", "non", "ee", "ee", "ee"), "col2"=c(1:6, 2:5, 1:6))

print(intersect_mod(datf=datf, inter_col=2, mod_col=1, n_min=3))

  col1 col2
8   non   3
9   non   4
```

```

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

```

inter_max

inter_max

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.

Examples

```

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

```

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 algorithm 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:

Examples

```
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]]
```



```
# [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
```

isnt_divisible	<i>isnt_divisible</i>
----------------	-----------------------

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

inpt_v	is the input vector
divisible_v	is the vector containing all the numbers that will try to divide those contained in inpt_v

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	<i>is_divisible</i>
--------------	---------------------

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

inpt_v	is the input vector
divisible_v	is the vector containing all the numbers that will try to divide those contained in inpt_v

Examples

```
print(is_divisible(inpt_v=c(1:111), divisible_v=c(2, 4, 5)))

#[1] 20 40 60 80 100
```

join_n_lvl	<i>join_n_lvl</i>
------------	-------------------

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

- frst_datf 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
- lst_pair 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

```
datf3 <- data.frame("vil"=c("one", "one", "one", "two", "two", "two"),
                    "charac"=c(1, 2, 2, 1, 2, 2),
                    "rev"=c(1250, 1430, 970, 1630, 2231, 1875),
                    "vil2" = c("one", "one", "one", "two", "two", "two"),
                    "idl2" = c(1:6))
datf4 <- data.frame("vil"=c("one", "one", "one", "two", "two", "three"),
                    "charac"=c(1, 2, 2, 1, 1, 2),
                    "rev"=c(1.250, 1430, 970, 1630, 593, 456),
                    "vil2" = c("one", "one", "one", "two", "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("vil2" = "idl2")),
                join_type=c("inner", "left"))

[1] "pair: charac vil"
| | 0%
1
|= | 50%
2
|==| 100%
[1] "pair: vil2 idl2"
```

```
| | 0%
one
|= | 50%
two
|==| 100%

main_id.x vil.x charac.x rev.x vil2.x idl2.x main_id.y vil.y charac.y rev.y
1 1oneone1 one 1 1250 one 1 <NA> <NA> NA NA
2 2oneone2 one 2 1430 one 2 <NA> <NA> NA NA
3 2oneone3 one 2 970 one 3 2oneone3 one 2 1430
4 1twotwo4 two 1 1630 two 4 <NA> <NA> NA NA
vil2.y idl2.y
1 <NA> NA
2 <NA> NA
3 one 3
4 <NA> NA
```

just_anything	<i>just_anything</i>
---------------	----------------------

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	<i>just_anything2</i>
----------------	-----------------------

Description

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

Usage

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

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	<i>just_anything3</i>
----------------	-----------------------

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] "oui jj" "oui jj"
```

just_chr	<i>just_chr</i>
----------	-----------------

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

Examples

```
print(just_chr(inpt_v = c("oui222jj644", "oui122jj"),
  symbol_ = "-"))

[1] "oui-jj-" "oui-jj"
```

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

```
print(just_chr2(inpt_v = c("oui222jj44", "oui122jj"),
  symbol_ = "-"))

[1] "oui---jj--" "oui---jj"
```

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

Examples

```
print(just_chr3(inpt_v = c("oui222jj644", "oui122jj")))

[1] "oui jj" "oui jj"
```

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

```
print(just_nb(inpt_v = c("oui222jj644", "oui122jj"),
  symbol_ = "-"))

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

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

Examples

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

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

just_nb3	<i>just_nb3</i>
----------	-----------------

Description

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

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	<i>just_not_anything</i>
-------------------	--------------------------

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

Examples

```
print(just_not_anything(inpt_v = c("oui222jj644", "oui122jj"),
  symbol_ = "-", anything_v = letters))
[1] "-222-644" "-122-"
```

`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

<code>inpt_v</code>	is the input vector
<code>symbol_</code>	is the chosen symbol to replace numbers

Examples

```
print(just_not_anything2(inpt_v = c("oui222jj44", "oui122jj"),  
  symbol_ = "-", anything_v = letters))
```

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

`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

<code>inpt_v</code>	is the input vector
---------------------	---------------------

Examples

```
print(just_not_anything3(inpt_v = c("oui222jj644", "oui122jj"),  
  anything_v = letters))
```

```
[1] "222644" "122"
```

leap_yr	<i>leap_year</i>
---------	------------------

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	<i>left_all</i>
----------	-----------------

Description

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

Usage

```
left_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

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(left_all(datf1, datf2, datf2, datf2, keep_val=FALSE, id_v="idl"))

  idl var1.x var1.y var1.x.x var1.y.y
1   1   oui  oui2   oui2   oui2
2   2   oui  oui2   oui2   oui2
3   3   oui  oui2   oui2   oui2
4   4  non  <NA>   <NA>   <NA>
5   5  non  <NA>   <NA>   <NA>#'
print(left_all(datf1, datf2, datf2, keep_val=FALSE, id_v="idl"))

  idl var1.x var1.y var1
1   1   oui  oui2 oui2
2   2   oui  oui2 oui2
3   3   oui  oui2 oui2
4   4  non  <NA> <NA>
5   5  non  <NA> <NA>
```

letter_to_nb	<i>letter_to_nb</i>
--------------	---------------------

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	<i>list_files</i>
------------	-------------------

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 = ".")
```

Arguments

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

lst_flatnr	<i>lst_flatnr</i>
------------	-------------------

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	<i>match_by</i>
----------	-----------------

Description

Allow to match elements by ids, see examples.

Usage

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

Arguments

to_match_v is the vector containing all the elements to match
 inpt_v is the input vector containong 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

Examples

```
print(match_by(to_match_v = c("a"), inpt_v = c("a", "z", "a", "p", "p", "e", "e", "a"),
             inpt_ids = c(1, 1, 1, 2, 2, 3, 3, 3)))

[1] 1 8

print(match_by(to_match_v = c("a"), inpt_v = c("a", "z", "a", "a", "p", "e", "e", "a"),
             inpt_ids = c(1, 1, 1, 2, 2, 3, 3, 3)))

[1] 1 4 8

print(match_by(to_match_v = c("a", "e"), inpt_v = c("a", "z", "a", "a", "p", "e", "e", "a"),
             inpt_ids = c(1, 1, 1, 2, 2, 3, 3, 3)))

[1] 1 4 8 6
```

multitud	<i>multitud</i>
----------	-----------------

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_ = "")
```

Arguments

- `l` 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"
```

nb2_follow	<i>nb2_follow</i>
------------	-------------------

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

<code>inpt_v</code>	is the input vector
<code>inpt_idx</code>	is the index
<code>inpt_follow_v</code>	is a vector containing the patterns that are potentially just after <code>inpt_nb</code>

Examples

```
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_follow_v = c(5)))

[1] "3"    "oui"

# we have 3 times continuously the pattern 'oui' and 0 times the pattern 5 just after the 2nd index of inpt_v

print(nb2_follow(inpt_v = c(1, "non", "5", "5", "5", "nop", 5), inpt_idx = 2, inpt_follow_v = c(5)))

[1] "3" "5"
```

nb_follow	<i>nb_follow</i>
-----------	------------------

Description

Allow to get the number of certain 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

```
print(nb_follow(inpt_v = c(1:13), inpt_idx = 6, inpt_follow_v = c(5:9)))

[1] 3

print(nb_follow(inpt_v = c("ou", "nn", "pp", "zz", "zz", "ee", "pp"), inpt_idx = 2,
  inpt_follow_v = c("pp", "zz")))

[1] 3
```

<code>nb_to_letter</code>	<i>nb_to_letter</i>
---------------------------	---------------------

Description

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

Usage

```
nb_to_letter(x)
```

Arguments

`x` is the number of the column

Examples

```
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] "zxx"

print(nb_to_letter(18277))

[1] "zzy"

print(nb_to_letter(18278))

[1] "zzz"

print(nb_to_letter(18279))

[1] "aaaa"

```

nestr_datfl

nestr_datfl

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

`inptf_datf` is the input dataframe (1b)

`inptt_pos_datf` is the dataframe (2b) that corresponds to the (1a) values

`nestr_datf` is the dataframe (2b) that has the special value (2a)

`yes_val` is the special value (2a)

`inptt_neg_datf` is the dataframe (4b) that has the (3a) values, defaults to NA

Examples

```
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
#2        44         12
#3         3          2

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.
#1         4          1
#2         2          5
#3         3          2
```

nestr_datf2

nestr_datf2

Description

Allow to write a special value (1a) in the cells of a dataframe (1b) that correspond (row and column) to those 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

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

Examples

```
print(nestr_datf2(inptf_datf=data.frame(c(1, 2, 1), c(1, 5, 7)), rtn_pos="yes",
rtn_neg="no", nestr_datf=data.frame(c(TRUE, FALSE, TRUE), c(FALSE, FALSE, TRUE)), yes_val=T)

# c.1..2..1. c.1..5..7.
#1          yes          no
#2          no           no
#3          yes          yes
```

nest_v	nest_v
--------	--------

Description

Nest two vectors according to the following parameters.

Usage

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

Arguments

- f_v is the vector that will welcome the nested vector t_v
- t_v is the imbricator 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 begining 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"
```

new_ordered	<i>new_ordered</i>
-------------	--------------------

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

Examples

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

normal_dens	<i>normal_dens</i>
-------------	--------------------

Description

Calculates the normal distribution probability, see examples

Usage

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

Arguments

target_v	is the target value(s) (one or bounded), see examples
mean	is the mean of the normal distribution
sd	is the standard deviation of the normal distribution

Examples

```
print(normal_dens(target_v = 13, mean = 12, sd = 2))

[1] 0.1760327

print(normal_dens(target_v = c(9, 11), mean = 12, sd = 1.5, step = 0.01))

[1] 0.2288579

print(normal_dens(target_v = c(1, 18), mean = 12, sd = 1.5, step = 0.01))

[1] 0.9999688
```

<code>occu</code>	<i>occu</i>
-------------------	-------------

Description

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

Usage

```
occu(inpt_v)
```

Arguments

`inpt_v` the input dataframe

Examples

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

#   var occurrence
#1  oui           2
#2  peut          2
#3  non           1
```

<code>old_to_new_idx</code>	<i>old_to_new_idx</i>
-----------------------------	-----------------------

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

Examples

```
print(old_to_new_idx(inpt_v = c("oui", "no", "eeee")))

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

<code>pairs_findr</code>	<i>pairs_findr</i>
--------------------------	--------------------

Description

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

Usage

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

Arguments

`inpt` is the input character
`ptrn1` is the first pattern encountered 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
```

<code>pairs_findr_merger</code>	<i>pairs_findr_merger</i>
---------------------------------	---------------------------

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

Examples

```
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
```

```

[[2]]
[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)),
                        lst2 = list(c(3, 2, 1, 1, 2, 3, 4, 4), c(1, 17, 18, 22, 23, 29,
[[1]]
[1] 6 5 1 1 2 2 3 3 4 4 5 6 7 7

[[2]]
[1] 1 2 7 9 10 11 15 17 18 22 23 29 35 40

print(pairs_findr_merger(lst1 = list(c(1, 1), c(22, 23)),
                        lst2 = list(c(1, 1, 2, 2), c(3, 21, 27, 32))))

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

[[2]]
[1] 3 21 22 23 27 32

```

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 conjunction 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 prioritizing 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 algorithms used. The first algorithm 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 algorithm

will put parenthesis at the condition that are located after other conditions that are surrounded by a pair. The third algorithm 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

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)) "
```

pairs_insertr2

pairs_insertr2

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 conjunction 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("(", "["),
  flagged_conj_v = c("&", "|"),
  method = c("(", ")")
)

```

Arguments

<code>inpt</code>	is the input character representing an arbitrary condition, like ReGex for example, or information to a parser for example
<code>algo_used</code>	is a vector containing one or more of the 3 algorithms used. The first algorithm will simply put the pair of parenthesis at the condition surrounded and/or after a character flagged (in <code>flagged_conj_v</code>) as a conjunction. The second algorithm will put parenthesis at the condition that are located after other conditions that are surrounded by a pair. The third algorithm 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 algorithm(s) is the most efficient for your use case.
<code>flagged_pair_v</code>	is a vector containing all the first character of the pairs
<code>corr_v</code>	is a vector containing all the last character of the pairs
<code>flagged_conj_v</code>	is a vector containing all the conjunction character
<code>method</code>	is length 2 vector containing as a first index, the first character of the pair inserted, and at the last index, the second and last character of the pair

Examples

```

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), meth
      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

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 ""

Examples

```
print(paste_datf(inpt_datf=data.frame(c(1, 2, 1), c(33, 22, 55))))

[1] "133" "222" "155"
```

paste_datf2

paste_datf2

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 ""

Examples

```
print(paste_datf2(inpt_datf=data.frame(c(1, 2, 1), c(33, 22, 55))))

#[1] "121" "332255"
```

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, sep=" "))

# [1] "er-re-o-u-i" "ere-re-o-u-i" "ere-er-o-u-i"
```

pattern_gettr *pattern_gettr*

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 being 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

Usage

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

Arguments

<code>word_</code>	is the vector containing the patterns
<code>vct</code>	is the vector being searched for patterns
<code>occ</code>	a vector containing the occurrence of the pattern in <code>word_</code> to be matched in the vector being searched, if the occurrence is 2 for the nth pattern in <code>word_</code> and only one occurrence is found in <code>vct</code> so no pattern will be matched, put "forever" to no longer depend on the occurrence for the associated pattern
<code>strict</code>	a vector containing the "strict" condition for each nth vector in <code>word_</code> ("strict" is the string to activate this option)
<code>btwn</code>	is a vector containing the condition ("yes" to activate this option) meaning that if "yes", all elements between two matched pattern in <code>vct</code> will be returned , so the patterns you enter in <code>word_</code> have to be in the order you think it will appear in <code>vct</code>
<code>all_in_word</code>	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
<code>notatall</code>	is a string that you are sure is not present in <code>vct</code>

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"
```

pattern_tuning

pattern_tuning

Description

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

Usage

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

Arguments

patrn	is the character that will be tuned
spe_nb	is the number of new character that will be replaced
spe_l	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(patrnr="oui", spe_nb=2, spe_l=c("e", "r", "T", "O"), exclude_type="c")
# [1] "orT" "oTr" "oOi"
```

power_to_char	<i>power_to_char</i>
---------------	----------------------

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
--------	--

Examples

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

```
pre_to_post_idx      pre_to_post_idx
```

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 swiched 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.

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

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"
```

read_edm_parser	<i>read_edm_parser</i>
-----------------	------------------------

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

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))", to_find_v = c("ok", "oui", "rr", "rr2")))

[1] "56"

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] "56"
```

rearangr_v	<i>rearangr_v</i>
------------	-------------------

Description

Rearranges 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 rearranges 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 descendigly

Examples

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

regex_spe_detect	<i>regex_spe_detect</i>
------------------	-------------------------

Description

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

Usage

```
regex_spe_detect(inpt)
```

Arguments

inpt	the input character
------	---------------------

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/Z1, X1/Y1/Z2, ...)"

Usage

```

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

Arguments

<code>inpt_v</code>	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 "/"
<code>sep_</code>	is the unique separator separating the sub-elements in each elements of <code>inpt_v</code>
<code>order</code>	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 <code>order=c(3:1)</code> , for example, and it should returns something like this <code>c(X1/Y1/Z1, X2/Y1/Z1, X1/Y2/Z1, ...)</code> assuming you have only two values for X.
<code>l_order</code>	is a list containing the vectors of values you want to order first for each sub-elements

Examples

```
vec <- multitud(l=list(c("a", "b"), c("1", "2"), c("A", "Z", "E"), c("Q", "F")), sep_="/")

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" "a/2/9/9" "a/2/10/10" "a/2/11/11" "a/2/12/12"
# [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_="/"))

# [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" "a/2/9/9" "a/2/10/10" "a/2/11/11" "a/2/12/12"
# [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)))

# [1] "1/1/A/Q" "2/2/A/Q" "3/3/A/Q" "4/4/A/Q" "5/5/Z/Q" "6/6/Z/Q"
# [7] "7/7/Z/Q" "8/8/Z/Q" "9/9/E/Q" "10/10/E/Q" "11/11/E/Q" "12/12/E/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

Examples

```
datf <- data.frame(c(1, 2, NA, 4), c(1:4))
print(datf)
```

```
  c.1..2..NA..4. c.1.4.
1              1      1
2              2      2
3             NA      3
4              4      4
```

```
print(rm_na_rows(inpt_datf = datf))
```

```
  c.1..2..NA..4. c.1.4.
1              1      1
2              2      2
4              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

Examples

```
datf <- data.frame(c(1, 2, NA, 4), c(1:4))
print(datf)
```

```
  c.1..2..NA..4. c.1.4.
1              1      1
2              2      2
3             NA      3
4              4      4
```

```
print(rm_rows(inpt_datf = datf, flagged_vals = c(1, 4)))
```

```
  c.1..2..NA..4. c.1.4.
2              2      2
3             NA      3
```

row_to_col	<i>row_to_col</i>
------------	-------------------

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 <- 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         4
2        11        10         9         8

print(row_to_col(inpt_datf = datf_test))

  1  2
1 1 11
2 2 10
3 3  9
4 4  8
```

r_print	<i>r_print</i>
---------	----------------

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
`begn` is the character put at the beginning of the print
`end` is the character put at the end of the print

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

`inpt_l` is the input list containing all the vectors

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

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

see_datf

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

<code>datf</code>	is the input dataframe
<code>condition_l</code>	is the vector of the possible conditions (" <code>==</code> ", " <code>></code> ", " <code><</code> ", " <code>!=</code> ", " <code>%%</code> ", " <code>reg</code> ", " <code>not_reg</code> ", " <code>sup_nchar</code> ", " <code>inf_nchar</code> ", " <code>nchar</code> ") (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.
<code>val_l</code>	is the list of vectors containing the values or vector of values related to <code>condition_l</code> (so the vector of values has to be placed in the same order)
<code>conjunction_l</code>	contains the and or conjunctions, so if the length of <code>condition_l</code> is equal to 3, there will be 2 conjunctions. If the length of <code>conjunction_l</code> is inferior to the length of <code>condition_l</code> minus 1, <code>conjunction_l</code> will match its goal length value with its last argument as the last arguments. For example, <code>c("&", "l", "&")</code> with a goal length value of 5 -> <code>c("&", "l", "&", "&", "&")</code>
<code>rt_val</code>	is a special value cell returned when the conditions are respected
<code>f_val</code>	is a special value cell returned when the conditions are not respected

Details

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

Examples

```
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))))
```

```

#      X1      X2
#1 TRUE  TRUE
#2 TRUE  TRUE
#3 TRUE FALSE

print(see_datf(datf=datf1, condition_l=c("=="), val_l=list(c("a", 1))))

#      X1      X2
#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

Examples

```
print(see_diff(c(1:7), c(4:12)))

[1] 1 2 3 8 9 10 11 12
```

see_diff_all	<i>see_diff_all</i>
--------------	---------------------

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
```

see_file	<i>see_file</i>
----------	-----------------

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 occurrence 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"
```

see_idx

see_idx

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

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", "3-3-sheet_1-out.xlsx", 4-5-sheet_2-out.xlsx)

Usage

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

Arguments

- pattern_ is a vector containin the file extension of the spreadsheets ("xlsx", "csv"...)
- path_ is the path where are located the files
- 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.
- unique_sep is a pattern that you know will never be in your input files
- rec is a boolean allows to get files recursively if set to TRUE, defaults to TRUE If x 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	<i>see_in_grep</i>
-------------	--------------------

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

- from_v 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

```
print(see_in_grep(from_v = c("oui", "non", "peut"),  
  in_v = c("ou", "eu", "plm"))  
  
      ou      eu      plm  
      1      3      -1  
  
print(see_in_grep(from_v = c("oui", "non", "peut", "oui"),
```

```
in_v = c("ou", "eu", "plm"))

$ou
[1] 1 4

$eu
[1] 3

$plm
[1] -1
```

`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())
```

Arguments

<code>from_v</code>	is the vector that may contains elements that contains the same patterns that those in <code>in_v</code> , see examples
<code>in_v</code>	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

<code>inpt_v</code>	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	<i>selected_char</i>
---------------	----------------------

Description

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

Usage

```
selected_char(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 character is generated

Examples

```
print(selected_char(1222))

[1] "zta"
```

sequence_na_mean1	<i>sequence_na_mean1</i>
-------------------	--------------------------

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
-----------	------------------------

Examples

```

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_sequence1(inpt_datf = datf, bf_ = 2)
datf[3, 4] <- NA
datf[6, 4] <- NA
datf[1, 3] <- NA
print(datf)

```

	id_seq	individual	var1-1	var1-2	var2-1	var2-2
1	20	oui	NA	120	20	19
2	20	non	NA	112	NA	17
3	20	peut1	101	NA	14	17
4	20	peut2	112	121	17	20
5	19	oui	120	110	19	17
6	19	peut1	110	NA	17	18
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	20	oui	115	120.0	20	19
2	20	non	112	112.0	17	17
3	20	peut1	101	105.5	14	17
4	20	peut2	112	121.0	17	20
5	19	oui	120	110.0	19	17
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

Examples

```
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_sequence2(inpt_datf = datf, bf_ = 2)
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
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

```
print(sequence_na_mean2(inpt_datf = datf, bf_ = 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

`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

Examples

```
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_sequence1(inpt_datf = datf, bf_ = 2)
datf[3, 4] <- NA
datf[6, 4] <- NA
datf[1, 3] <- NA
print(datf)
```

	id_seq	individual	var1-1	var1-2	var2-1	var2-2
1	20	oui	NA	120	20	19
2	20	non	NA	112	NA	17
3	20	peut1	101	NA	14	17
4	20	peut2	112	121	17	20
5	19	oui	120	110	19	17
6	19	peut1	110	NA	17	18
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
1	20	oui	115	120.0	20	19
2	20	non	112	112.0	17	17
3	20	peut1	101	105.5	14	17
4	20	peut2	112	121.0	17	20
5	19	oui	120	110.0	19	17
6	19	peut1	110	105.5	17	18
7	19	peut2	121	113.0	20	17

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

Examples

```
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_sequence2(inpt_datf = datf, bf_ = 2)
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
1	20	oui	NA	121	120	16	20	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

```
print(sequence_na_med2(inpt_datf = datf, bf_ = 2))
```

	id_seq	individual	var1-0	var1-1	var1-2	var2-0	var2-1	var2-2
1	20	oui	120	121.0	120	16	20	19
2	20	non	117	114.5	112	19	18	17
3	20	peut1	109	109.0	110	16	14	17
4	20	peut2	119	112.0	121	19	17	20
5	19	oui	121	120.0	110	20	19	17
6	19	peut1	101	109.0	115	14	17	18
7	19	peut2	112	121.0	113	17	20	17

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

```
print(sort_date(inpt_v = c("01-11-2025", "08-08-1922", "12-04-1966")
, frmt = "dmy", sep_ = "-", ascending = TRUE, give = "value"))

[1] "08-08-1922" "12-04-1966" "01-11-2025"

print(sort_date(inpt_v = c("01-11-2025", "08-08-1922", "12-04-1966")
, frmt = "dmy", sep_ = "-", ascending = FALSE, give = "value"))

[1] "01-11-2025" "12-04-1966" "08-08-1922"

print(sort_date(inpt_v = c("01-11-2025", "08-08-1922", "12-04-1966")
, frmt = "dmy", sep_ = "-", ascending = TRUE, give = "index"))

[1] 2 3 1

print(sort_date(inpt_v = c("22-01-11-2025", "11-12-04-1966", "12-12-04-1966")
, frmt = "hdmy", sep_ = "-", ascending = FALSE, give = "value"))

[1] "22-01-11-2025" "12-12-04-1966" "11-12-04-1966"

print(sort_date(inpt_v = c("03-22-01-11-2025", "56-11-12-04-1966", "23-12-12-04-1966")
, frmt = "nhdmy", sep_ = "-", ascending = FALSE, give = "value"))

[1] "03-22-01-11-2025" "23-12-12-04-1966" "56-11-12-04-1966"
```

```
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 algorith 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

Examples

```
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

print(datf_test)
```

	sample_qual	sample_freq
1	a	0.208695652
2	b	0.234782609
3	c	0.321739130
4	d	0.339130435
5	e	0.330434783
6	f	0.069565217
7	g	0.234782609
8	h	0.400000000
9	i	0.347826087
10	j	0.043478261
11	k	0.278260870
12	l	0.286956522
13	m	0.243478261
14	n	0.147826087
15	o	0.234782609
16	p	0.252173913
17	q	0.417391304
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

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	0.173913043
34	ah	0.278260870
35	ai	0.278260870
36	aj	0.347826087
37	ak	0.026086957
38	al	0.295652174
39	am	0.226086957
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	at	0.252173913
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
60	bh	0.043478261
61	bi	0.200000000
62	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	by	0.243478261
78	bz	0.034782609
79	ca	0.017391304
80	cb	0.008695652
81	cc	0.173913043
82	cd	0.147826087

```

83      ce 0.060869565
84      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      cp 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
102     cx 0.017391304
103     cy 0.043478261
104     cz 0.008695652
105     da 0.034782609
106     db 0.017391304
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
115     dk 0.008695652

```

```
print(sort_normal_qual(inpt_datf = datf_test))
```

```

0.00869565217391304 0.00869565217391304 0.00869565217391304 0.00869565217391304
      "aa"          "cb"          "cz"          "de"
0.00869565217391304 0.00869565217391304 0.0173913043478261 0.0173913043478261
      "dh"          "dk"          "bp"          "ca"
0.0173913043478261 0.0173913043478261 0.0173913043478261 0.0173913043478261
      "cl"          "cp"          "ct"          "db"
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
      "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"          "o"
0.234782608695652 0.243478260869565 0.243478260869565 0.252173913043478

```

"aq"	"m"	"by"	"at"
0.278260869565217	0.278260869565217	0.28695652173913	0.295652173913043
"k"	"ai"	"l"	"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	0.00869565217391304	
"ch"	"bl"	"t"	

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 algorithm 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

Examples

```

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])

print(datf_test)

```

```

      sample_qual sample_freq
1              a 0.208695652
2              b 0.234782609
3              c 0.321739130
4              d 0.339130435
5              e 0.330434783
6              f 0.069565217
7              g 0.234782609
8              h 0.400000000
9              i 0.347826087
10             j 0.043478261
11             k 0.278260870
12             l 0.286956522
13             m 0.243478261
14             n 0.147826087
15             o 0.234782609
16             p 0.252173913
17             q 0.417391304
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
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 0.173913043
34            ah 0.278260870
35            ai 0.278260870
36            aj 0.347826087
37            ak 0.026086957
38            al 0.295652174
39            am 0.226086957
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            at 0.252173913

```

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
60	bh	0.043478261
61	bi	0.200000000
62	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	by	0.243478261
78	bz	0.034782609
79	ca	0.017391304
80	cb	0.008695652
81	cc	0.173913043
82	cd	0.147826087
83	ce	0.060869565
84	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	cp	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
102	cx	0.017391304
103	cy	0.043478261

```

104      cz 0.008695652
105      da 0.034782609
106      db 0.017391304
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
115      dk 0.008695652

```

```
print(sort_normal_qual2(inpt_datf = datf_test))
```

```

0.00869565217391304 0.00869565217391304 0.00869565217391304 0.00869565217391304
      "aa"      "cb"      "cz"      "de"
0.00869565217391304 0.00869565217391304 0.0173913043478261 0.0173913043478261
      "dh"      "dk"      "bp"      "ca"
0.0173913043478261 0.0173913043478261 0.0173913043478261 0.0173913043478261
      "cl"      "cp"      "ct"      "db"
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
      "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"      "o"
0.234782608695652 0.243478260869565 0.243478260869565 0.252173913043478
      "aq"      "m"      "by"      "at"
0.278260869565217 0.278260869565217 0.28695652173913 0.295652173913043
      "k"      "ai"      "l"      "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	0.00869565217391304	
"ch"	"bl"	"t"	

split_by_step

*split_by_step***Description**

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

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_until	<i>str_remove_until</i>
------------------	-------------------------

Description

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

Usage

```
str_remove_until(
  inpt_v,
  ptrn_rm_v = c(),
  until = list(c(1)),
  nvr_following_ptrn = "NA"
)
```

Arguments

inpt_v	is the input vector
ptrn_rm_v	is a vector containing the patterns to remove
until	is a list containing the occurrence(s) of each pattern to remove in the elements.
nvr_following_ptrn	is a sequel of characters that you are sure is not present in any of the elements in inpt_v

Examples

```
vec <- c("45/56-/98mm", "45/56-/98mm", "45/56-/98-mm//")

print(str_remove_until(inpt_v=vec, ptrn_rm_v=c("-", "/"), until=list(c("max"), c(1))))

#[1] "4556/98mm"      "4556/98mm"      "4556/98mm//"

print(str_remove_until(inpt_v=vec, ptrn_rm_v=c("-", "/"), until=list(c("max"), c(1:2))))

#[1] "455698mm"       "455698mm"       "455698mm//"

print(str_remove_until(inpt_v=vec[1], ptrn_rm_v=c("-", "/"), until=c("max")))

#[1] "455698mm" "455698mm" "455698mm"
```

sub_mult	<i>sub_mult</i>
----------	-----------------

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 substitute those provided by `pattern_v`

Examples

```
print(sub_mult(inpt_v = c("X and Y programming languages are great", "More X, more X!"),
               pattern_v = c("X", "Y", "Z"),
               replacement_v = c("C", "R", "GO")))

[1] "C and R programming languages are great"
[2] "More C, more X!"
```

successive_diff	<i>successive_diff</i>
-----------------	------------------------

Description

Allow to see the difference between the successive 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 1 2 6
```

swipr	<i>swipr</i>
-------	--------------

Description

Returns an ordered dataframes according to the elements order given. The input dataframe 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

```
datf <- data.frame("col1"=c("Af", "Al", "Al", "Al", "Arg", "Arg", "Arg", "Arm", "Arm", "A  

                    "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	Af	B
2	Al	G
3	Al	S
4	Al	B
5	Arg	G
6	Arg	S
7	Arg	B
8	Arm	G
9	Arm	B
10	Al	B

test_order	<i>test_order</i>
------------	-------------------

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
```

<code>to_unique</code>	<i>to_unique</i>
------------------------	------------------

Description

Allow to transform a vector containing elements that have more than 1 occurence to a vector with only unques elements.

Usage

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

Arguments

`inpt_v` is the input vectors

`distinct_type` takes two values: suffix or prefix

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

Examples

```
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"
```

<code>union_all</code>	<i>union_all</i>
------------------------	------------------

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
```

<code>union_keep</code>	<i>union_keep</i>
-------------------------	-------------------

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

Examples

```
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	<i>unique_datf</i>
-------------	--------------------

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

Examples

```
datf1 <- data.frame(c(1, 2, 1, 3), c("a", "z", "a", "p"))

print(datf1)

  c.1..2..1..3. c..a....z....a....p.. c.1..2..1..3..1
1              1                      a              1
2              2                      z              2
3              1                      a              1
4              3                      p              3

print(unique_datf(inpt_datf=datf1))

#   c.1..2..1..3. c..a....z....a....p..
#1              1                      a
#2              2                      z
#4              3                      p

datf1 <- 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          a
2          2          z
3          1          a
4          3          p

print(unique_datf(inpt_datf=datf1, col=TRUE))

#  cur_v cur_v
#1     1     a
#2     2     z
#3     1     a
#4     3     p
```

unique_ltr_from_v *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

Examples

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

unique_total	<i>unique_total</i>
--------------	---------------------

Description

Returns a vector with the total amount of occurrences for each element in the input vector. The occurrences 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 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 1 3 1
```

until_stnl	<i>until_stnl</i>
------------	-------------------

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

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
#[39] 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3
```

val_replacer	<i>val_replacer</i>
--------------	---------------------

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

```
print(val_replacer(datf=data.frame(c(1, "oo4", TRUE, FALSE), c(TRUE, FALSE, TRUE, TRUE)),
  val_replaced=c(TRUE), val_replacor="NA"))

# c.1...oo4...T..F. c.T..F..T..T.
#1                1                NA
#2                oo4              FALSE
#3                NA                NA
#4                FALSE              NA
```

vector_replacor	<i>vector_replacor</i>
-----------------	------------------------

Description

Allow to replace certain values in a vector.

Usage

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

Arguments

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

Examples

```
print(vector_replacor(inpt_v=c(1:15), sus_val=c(3, 6, 8, 12),
  rpl_val=c("oui", "non", "e", "a")))

# [1] "1" "2" "oui" "4" "5" "non" "7" "e" "9" "10" "11" "a"
#[13] "13" "14" "15"

print(vector_replacor(inpt_v=c("non", "zez", "pp a ftf", "fdatfd", "assistance",
  "ert", "repas", "repos"),
  sus_val=c("pp", "as", "re"), rpl_val=c("oui", "non", "zz"), grep_=TRUE))

#[1] "non" "zez" "oui" "fdatfd" "non" "ert" "non" "zz"
```

vec_in_datf	<i>vec_in_datf</i>
-------------	--------------------

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_until` 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

Examples

```

datf1 <- data.frame(c(1:5), c(5:1), c("a", "z", "z", "z", "a"))

print(datf1)

#  c.1.5. c.5.1. c..a....z....z....z....a..
#1      1      5                                a
#2      2      4                                z
#3      3      3                                z
#4      4      2                                z
#5      5      1                                a

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	<i>vlookup_datf</i>
--------------	---------------------

Description

Allow to perform a vlookup on a dataframe

Usage

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

Arguments

datf	is the input dataframe
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))

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                                az2      4      1
#21                               az3      2      3
#3                                az4      3      2
```

wider_datf	<i>wider_datf</i>
------------	-------------------

Description

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

Usage

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

Arguments

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

Examples

```

datf1 <- data.frame(c(1:5), c("o-y", "hj-yy", "er-y", "k-ll", "ooo-mm"), c(5:1))

datf2 <- 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.
#o-y    1      "o"   "y"   5
#hj-yy  2      "hj"  "yy"  4
#er-y   3      "er"  "y"   3
#k-ll   4      "k"   "ll"  2
#ooo-mm 5      "ooo" "mm"  1

print(wider_datf(inpt_datf=datf2, col_to_splt=c("col2"), sep="-"))

#      pre_datf X.o.  X.y.
#o-y    1      "o"   "y"
#hj-yy  2      "hj"  "yy"
#er-y   3      "er"  "y"
#k-ll   4      "k"   "ll"
#ooo-mm 5      "ooo" "mm"

```

wide_to_narrow_idx *wide_to_narrow_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

Examples

```

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

[1] 2 4 5

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

[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	<i>write_edm_parser</i>
------------------	-------------------------

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

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_stat, 5
any_join_datf, 6
appndr, 8
arroundr_mean, 9
arroundr_min, 9

better_match, 10
better_split, 11
better_split_any, 11
better_sub, 12
better_sub_mult, 13
better_unique, 14
bind_cols, 15
bind_rows, 16

can_be_num, 16
closer_ptrn, 17
closer_ptrn_adv, 19
clusterizer_v, 21
col_to_row, 24
colins_datf, 23
converter_date, 25
converter_format, 26
cost_and_taxes, 26
cumulated_rows, 27
cumulated_rows_na, 28
cut_v, 29
cutr_v, 29

data_gen, 30
data_meshup, 32
date_addr, 33
date_converter_reverse, 34
datf_appendr, 35
datf_appendr2, 35
datf_insertr, 36
datf_row_appendr, 37
datf_row_appendr2, 37
dcr_until, 38
dcr_val, 39
depth_pairs_findr, 39
diff_datf, 40
dynamic_idx_convertr, 40
edm_group_by1, 41
edm_group_by2, 42
edm_pivot_longer1, 43
edm_pivot_longer2, 45
edm_pivot_wider1, 47
edm_pivot_wider2, 49
elements_equalifier, 50
equalizer_v, 51
extract_normal, 52
extrt_only_v, 58

fillr, 58
fixer_nest_v, 59
fold_rec, 60
fold_rec2, 60
format_date, 61

geo_min, 61
get_rec, 62
globe, 62
glue_groupr_v, 63
grep_all, 64
grep_all2, 64
groupr_datf, 65
gsub_mult, 66

historic_sequence1, 67
historic_sequence2, 68
how_normal, 69
how_unif, 72

id_keepr, 73
incr_fillr, 74
infinite_char_seq, 75
inner_all, 75
insert_datf, 76
inter_max, 79
inter_min, 80
intersect_all, 77
intersect_mod, 77
is_divisible, 81
isnt_divisible, 81

join_n_lvl, 82
just_anything, 83
just_anything2, 83

just_anything3, 84
just_chr, 84
just_chr2, 85
just_chr3, 85
just_nb, 86
just_nb2, 86
just_nb3, 87
just_not_anything, 87
just_not_anything2, 88
just_not_anything3, 88

leap_yr, 89
left_all, 89
letter_to_nb, 90
list_files, 90
lst_flatnr, 91

match_by, 91
multitud, 92

nb2_follow, 93
nb_follow, 93
nb_to_letter, 94
nest_v, 97
nestr_datf1, 95
nestr_datf2, 96
new_ordered, 98
normal_dens, 98

occu, 99
old_to_new_idx, 99

pairs_findr, 100
pairs_findr_merger, 100
pairs_insertr, 102
pairs_insertr2, 103
paste_datf, 105
paste_datf2, 105
pattern_generator, 106
pattern_gettr, 106
pattern_tuning, 107
power_to_char, 108
pre_to_post_idx, 109
ptrn_switchr, 109
ptrn_twkr, 110

r_print, 116
read_edm_parser, 111
rearangr_v, 112
regex_spe_detect, 112
regroupr, 113
rm_na_rows, 114
rm_rows, 115
row_to_col, 116

save_until, 117
see_datf, 118
see_diff, 119
see_diff_all, 120
see_file, 120
see_idx, 121
see_in_grep, 122
see_in_l, 123
see_inside, 121
see_mode, 123
selected_char, 124
sequence_na_mean1, 124
sequence_na_mean2, 125
sequence_na_med1, 126
sequence_na_med2, 127
sort_date, 129
sort_normal_qual, 130
sort_normal_qual2, 133
split_by_step, 137
str_remove_until, 138
sub_mult, 138
successive_diff, 139
swipr, 140

test_order, 140
to_unique, 141

union_all, 142
union_keep, 142
unique_datf, 143
unique_ltr_from_v, 144
unique_pos, 144
unique_total, 145
until_stnl, 146

val_replacer, 146
vec_in_datf, 147
vector_replacor, 147
vlookup_datf, 149

wide_to_narrow_idx, 150
wider_datf, 149
write_edm_parser, 151