

# Package ‘edm1’

June 10, 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 functions 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,  
openxlsx

## Contents

all_stat . . . . .	3
any_join_datf . . . . .	5
appndr . . . . .	7
better_match . . . . .	8
better_split . . . . .	8
better_sub . . . . .	9
better_sub_mult . . . . .	10
can_be_num . . . . .	10
closer_ptrn . . . . .	11
closer_ptrn_adv . . . . .	13
clusterizer_v . . . . .	15
colins_datf . . . . .	17
converter_date . . . . .	18
converter_format . . . . .	19
cost_and_taxes . . . . .	20
cutr_v . . . . .	21
cut_v . . . . .	21
data_gen . . . . .	22
data_meshup . . . . .	24
date_addr . . . . .	25
date_converter_reverse . . . . .	26

dcr_untl . . . . .	27
dcr_val . . . . .	28
depth_pairs_findr . . . . .	28
diff_datf . . . . .	29
equalizer_v . . . . .	29
extrt_only_v . . . . .	30
fillr . . . . .	31
fixer_nest_v . . . . .	31
fold_rec . . . . .	32
fold_rec2 . . . . .	32
format_date . . . . .	33
geo_min . . . . .	33
get_rec . . . . .	34
globe . . . . .	34
grep_all . . . . .	35
groupr_datf . . . . .	36
gsub_mult . . . . .	37
id_keepr . . . . .	37
incr_fillr . . . . .	38
inner_all . . . . .	39
insert_datf . . . . .	40
intersect_all . . . . .	41
intersect_mod . . . . .	41
inter_max . . . . .	43
inter_min . . . . .	44
isnt_divisible . . . . .	45
is_divisible . . . . .	45
join_n_lvl . . . . .	46
leap_yr . . . . .	47
left_all . . . . .	47
letter_to_nb . . . . .	48
list_files . . . . .	49
lst_flatnr . . . . .	49
match_by . . . . .	50
multitud . . . . .	50
nb2_follow . . . . .	51
nb_follow . . . . .	52
nb_to_letter . . . . .	52
nestr_datf1 . . . . .	53
nestr_datf2 . . . . .	54
nest_v . . . . .	54
new_ordered . . . . .	55
non_unique . . . . .	56
occu . . . . .	56
pairs_findr . . . . .	57
pairs_findr_merger . . . . .	58
pairs_insertr . . . . .	59
pairs_insertr2 . . . . .	61
paste_datf . . . . .	62
pattern_generator . . . . .	63
pattern_gettr . . . . .	63
pattern_tuning . . . . .	64

power_to_char . . . . .	65
pre_to_post_idx . . . . .	66
ptrn_switchr . . . . .	66
ptrn_twkr . . . . .	67
rearangr_v . . . . .	68
regex_spe_detect . . . . .	69
regrouppr . . . . .	69
r_print . . . . .	70
save_untl . . . . .	71
see_datf . . . . .	72
see_diff . . . . .	73
see_diff_all . . . . .	74
see_file . . . . .	74
see_idx . . . . .	75
see_inside . . . . .	76
see_mode . . . . .	76
str_remove_untl . . . . .	77
sub_mult . . . . .	78
swipr . . . . .	78
union_all . . . . .	79
unique_datf . . . . .	80
unique_ltr_from_v . . . . .	81
unique_pos . . . . .	81
unique_total . . . . .	82
until_stnl . . . . .	82
val_replacer . . . . .	83
vector_replacor . . . . .	83
vec_in_datf . . . . .	84
vlookup_datf . . . . .	85
wider_datf . . . . .	86
<b>Index</b>	<b>88</b>

---

all\_stat

all\_stat

---

## Description

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

## Usage

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



---

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>
<code>rtn_col</code>	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 <code>c()</code>
<code>d_val</code>	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>
#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>
#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 "hmn". 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"
```

---

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\_sub

*better\_sub***Description**

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

**Usage**

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

**Arguments**

inpt\_v                is a vector containing all the elements that contains expressions to be substituted  
 pattern               is the expression that will be substituted  
 replacement          is the expression that will substitute pattern  
 untl\_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",  

               untl = 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",  

               untl = c(2, 3)))  

  

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

[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(),
    untl_v = c()
)
```

### Arguments

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

### 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"),
                      untl = c(1, 3)))

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

---

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 other 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

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

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

## 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
```

---

converter_date	converter_date
----------------	----------------

---

**Description**

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

**Usage**

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

**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
```

---

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

*v\_to\_datf*


---

### 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_distrib = 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", "v", "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 functio 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"
```

---

`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**

<code>strt_val</code>	is the start value
<code>cr_val</code>	is the incremental (or decremental value)
<code>stop_val</code>	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

*diff\_datf*


---

**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
```

---

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**

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

**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?????????"
```

---

<code>extrt_only_v</code>	<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**

<code>inpt_v</code>	is the input vector
<code>pttrn_v</code>	is the vector containing all the elements that can be in <code>inpt_v</code>

**Examples**

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

#[1] "oui" "oui" "oui" "oui"
```

---

fillr

*fillr*


---

### Description

Allow to fill a vector by the last element n times

### Usage

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

### Arguments

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

### Examples

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

---

fixer\_nest\_v

*fixer\_nest\_v*


---

### Description

Retur the elements of a vector "wrk\_v" (1) that corresponds to the pattern of elements in another vector "cur\_v" (2) according to another vector "pttrn\_v" (3) that contains the 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	<i>fold_rec</i>
----------	-----------------

---

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 algorith (which can be faster), check file\_rec2. Depth example: if i have dir/dir2/dir3, dir/dir2b/dir3b, i have a depth equal to 3

Usage

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

Arguments

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

---

fold_rec2	<i>fold_rec2</i>
-----------	------------------

---

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 algorith that does not nee to perform a full recursive search before tuning it to only find the directories with a good value of depth. Depth example: if i have dir/dir2/dir3, dir/dir2b/dir3b, i have a depth equal to 3

Usage

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



**Arguments**

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

---

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 first column is for latitude, the second for the longitude and the third, if exists, is for the altitude. Each point is one row.
established_datf	is the dataframe containing the coordinates of the established geographical points

**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,

#[1] 6342.844 7059.087
```

---

```
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
```

---

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

`inpt_datf` is the input dataframe

`condition_lst` is a list containing all the condition as a vector for each group

`val_lst` is a list containing all the values associated with `condition_lst` as a vector for each group

`conjunction_lst` is a list containing all the conjunctions associated with `condition_lst` and `val_lst` as a vector for each group

`rtn_val_pos` is a vector containing all the group flag value like this ex: `c("flag1", "flag2", "flag3")`

## 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("%%"), 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!"
```

---

id\_keepr

id\_keepr\_datf

---

### 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**

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

**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**

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

**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"
```

---

inner\_all

---

inner\_all

---

**Description**

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

**Usage**

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

**Arguments**

<code>...</code>	are all the dataframes etc
<code>keep_val</code>	is if you want to keep the id column
<code>id_v</code>	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>edml 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**

<code>inpt_l</code>	is the input list containing all the intervals
<code>min_</code>	is a value you are sure is superior to the maximum step value in all the intervals
<code>sensi</code>	is the decimal accuracy of how the difference between each value $n$ to $n+1$ in an interval is calculated
<code>sensi2</code>	is the decimal accuracy of how the value with the common step is calculated in all the intervals
<code>how_to_op</code>	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.
<code>how_to_val</code>	is a vector containing the value relatives to the operations in <code>hot_to_op</code> , 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(first_datf, scd_datf, join_type = c(), lst_pair = list())
```

Arguments

- first\_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(first_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
```

---

leap_yr	<i>bsx_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="id1"))
```

	id1	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="id1"))
```

	id1	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

*letter\_to\_nb***Description**

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

**Usage**

```
letter_to_nb(letter)
```

**Arguments**

letter is the letter (name of the column)



**Examples**

```
print(letter_to_nb("rty"))

#[1] 12713
```

---

list_files	<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

*nb2\_follow*


---

**Description**

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

**Usage**

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

**Arguments**

`inpt_v` is the input vector

`inpt_idx` is the index

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

**Examples**

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

[1] 1 5

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

[1] "3" "oui"
```

---

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

---

### Description

Allow to get the number of certain patterns that may be after an index of a vector, 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
```

---

nb_to_letter	<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(12713))

#[1] "rty"
```

---

nestr_datf1	<i>nestr_datf1</i>
-------------	--------------------

---

## 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
<hr/>		
nestr_datf2	<i>nestr_datf2</i>	
<hr/>		

**Description**

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

**Usage**

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

**Arguments**

- 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

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

nest_v	nest_v
<hr/>	

**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 imbriuator 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
```

---

non_unique	<i>non_unique</i>
------------	-------------------

---

**Description**

Returns the element that are not unique from the input vector

**Usage**

```
non_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 synthax is the following "comparaison\_type-actual\_value-". The comparaison type may be "==" or ">". Occu can also be a vector containing all the occurrence that must have the elements to be returned.

**Examples**

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

#[1] "oui" "non"

print(non_unique(inpt_v=c("oui", "oui", "non", "non", "peut", "peut1", "non"), occu=="==2-"))

#[1] "oui"

print(non_unique(inpt_v=c("oui", "oui", "non", "non", "peut", "peut1", "non"), occu=">-2-"))

#[1] "non"

print(non_unique(inpt_v=c("oui", "oui", "non", "non", "peut", "peut1", "non"), occu=c(1, 2)))

#[1] "non" "peut" "peut1"
```

---

occu	<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>pairs_findr</code>	<i><code>pairs_findr</code></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
```

---

pairs\_findr\_merger *pairs\_findr\_merger*

---

## Description

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

## Usage

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

## Arguments

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

## 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**

<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

**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	<i>pairs_insertr2</i>
----------------	-----------------------

---

## 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_insertr2(
  inpt,
  algo_used = c(1:3),
  flagged_pair_v = c(")", "["),
  corr_v = c("(", "["),
  flagged_conj_v = c("&", "|"),
  method = 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
method	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), method = "flagged_pair_v", 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 = "flagged_pair_v", 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(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"

```

---

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 authorize 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" "000000.18"
```

---

```
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"
```

---

rearangr\_v

---

*rearangr\_v*


---

**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 descendingly

**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      regex_spe_detect
```

---

### 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="fgfg[fggf]fgfg(vg?fgfgf.gf)"))
[1] "fgfg\\[fggf\\]fgfg\\(vg\\?fgfgf\\.gf\\)"
```

---

```
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

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

## 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"
```

---

r\_print

---

r\_print

---

## Description

Allow to print vector elements in one row.

**Usage**

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

**Arguments**

inpt_v	is the input vector
sep_	is the separator between each elements
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

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

---

### 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 associated with, do the following:

---

see_mode	<i>see_mode</i>
----------	-----------------

---

### Description

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

### Usage

```
see_mode(inpt_v = c())
```

### Arguments

inpt_v	is the input vector
--------	---------------------

**Examples**

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

[1] 2

print(see_mode(inpt_v = c(1, 1, 2, 2, 2, 3, 1)))

[1] 1
```

---

str_remove_untl	<i>str_remove_untl</i>
-----------------	------------------------

---

**Description**

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

**Usage**

```
str_remove_untl(
  inpt_v,
  ptrn_rm_v = c(),
  unt1 = 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
unt1	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_untl(inpt_v=vec, ptrn_rm_v=c("-", "/"), unt1=list(c("max"), c(1))))

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

print(str_remove_untl(inpt_v=vec, ptrn_rm_v=c("-", "/"), unt1=list(c("max"), c(1:2))))

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

print(str_remove_untl(inpt_v=vec[1], ptrn_rm_v=c("-", "/"), unt1=c("max")))

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

---

sub\_mult

*sub\_mult*


---

### Description

Performs a sub operation with n patterns and replacements.

### Usage

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

### Arguments

`inpt_v` is a vector containing all the elements that contains expressions to be substituted

`pattern_v` is a vector containing all the patterns to be substituted in any elements of `inpt_v`

`replacement_v` is a vector containing the expression that are going to 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!"
```

---

swipr

*swipr*


---

### 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"),
                   "col2"=c("B", "B", "G", "S", "B", "S", "G", "B", "G"))

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

datf <- data.frame("col1"=c("Af", "Arg", "Al", "Al", "Arg", "Arg", "Arg", "Arm", "Arm"),
                   "col2"=c("B", "B", "G", "S", "B", "S", "G", "B", "G"))

print(swipr(inpt_datf=datf, how_to=c("G", "S", "B"), id_w="col2", id_ids="col1"))

#   col1 col2
#1   Af    B
#2  Arg    G
#3   Al    G
#4   Al    S
#5  Arg    S
#6  Arg    B
#7  Arg    B
#8  Arm    G
#9  Arm    B

```

union\_all

*union\_all***Description**

Allow to perform a union function to n vectors.

**Usage**

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

**Arguments**

... are all the input vectors

**Examples**

```
print(union_all(c(1, 2), c(3, 4), c(1:8)))

[1] 1 2 3 4 5 6 7 8

print(union_all(c(1, 2), c(3, 4), c(7:8)))

[1] 1 2 3 4 7 8
```

---

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(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(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 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"
```

---

<code>vec_in_datf</code>	<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**

<code>inpt_datf</code>	is the input dataframe
<code>inpt_vec</code>	is the vector that may be in the input dataframe
<code>coeff</code>	is the "slope coefficient" of <code>inpt_vec</code>
<code>stop_untl</code>	is the maximum number of the input vector the function returns, if in the dataframe
<code>conventional</code>	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

*vlookup\_datf***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"
```

# Index

all\_stat, 3  
any\_join\_datf, 5  
appndr, 7  
  
better\_match, 8  
better\_split, 8  
better\_sub, 9  
better\_sub\_mult, 10  
  
can\_be\_num, 10  
closer\_ptrn, 11  
closer\_ptrn\_adv, 13  
clusterizer\_v, 15  
colins\_datf, 17  
converter\_date, 18  
converter\_format, 19  
cost\_and\_taxes, 20  
cut\_v, 21  
cutr\_v, 21  
  
data\_gen, 22  
data\_meshup, 24  
date\_addr, 25  
date\_converter\_reverse, 26  
dcr\_untl, 27  
dcr\_val, 28  
depth\_pairs\_findr, 28  
diff\_datf, 29  
  
equalizer\_v, 29  
extrt\_only\_v, 30  
  
fillr, 31  
fixer\_nest\_v, 31  
fold\_rec, 32  
fold\_rec2, 32  
format\_date, 33  
  
geo\_min, 33  
get\_rec, 34  
globe, 34  
grep\_all, 35  
groupr\_datf, 36  
gsub\_mult, 37  
  
id\_keepr, 37  
incr\_fillr, 38  
inner\_all, 39  
insert\_datf, 40  
inter\_max, 43  
inter\_min, 44  
intersect\_all, 41  
intersect\_mod, 41  
is\_divisible, 45  
isnt\_divisible, 45  
  
join\_n\_lvl, 46  
  
leap\_yr, 47  
left\_all, 47  
letter\_to\_nb, 48  
list\_files, 49  
lst\_flatnr, 49  
  
match\_by, 50  
multitud, 50  
  
nb2\_follow, 51  
nb\_follow, 52  
nb\_to\_letter, 52  
nest\_v, 54  
nestr\_datf1, 53  
nestr\_datf2, 54  
new\_ordered, 55  
non\_unique, 56  
  
occu, 56  
  
pairs\_findr, 57  
pairs\_findr\_merger, 58  
pairs\_insertr, 59  
pairs\_insertr2, 61  
paste\_datf, 62  
pattern\_generator, 63  
pattern\_gettr, 63  
pattern\_tuning, 64  
power\_to\_char, 65  
pre\_to\_post\_idx, 66  
ptrn\_switchr, 66  
ptrn\_twkr, 67



`r_print`, [70](#)  
`rearangr_v`, [68](#)  
`regex_spe_detect`, [69](#)  
`regroupr`, [69](#)  
  
`save_until`, [71](#)  
`see_datf`, [72](#)  
`see_diff`, [73](#)  
`see_diff_all`, [74](#)  
`see_file`, [74](#)  
`see_idx`, [75](#)  
`see_inside`, [76](#)  
`see_mode`, [76](#)  
`str_remove_until`, [77](#)  
`sub_mult`, [78](#)  
`swipr`, [78](#)  
  
`union_all`, [79](#)  
`unique_datf`, [80](#)  
`unique_ltr_from_v`, [81](#)  
`unique_pos`, [81](#)  
`unique_total`, [82](#)  
`until_stnl`, [82](#)  
  
`val_replacer`, [83](#)  
`vec_in_datf`, [84](#)  
`vector_replacor`, [83](#)  
`vlookup_datf`, [85](#)  
  
`wider_datf`, [86](#)