Package 'W4MRUtils'

September 8, 2023

Title Utils List for W4M - Workflow for Metabolomics

Version 1.0.0

Description Provides a set of utility function to prevent the spread of utilities script in W4M (Workflow4Metabolomics) scripts, and centralize them in a single package.
Some are meant to be replaced by real packages in a near future, like the parse_args() function: it is here only to prepare the ground for more global changes in W4M scripts and tools.

License AGPL (>= 3)

Encoding UTF-8

RoxygenNote 7.2.1

Imports methods, rlang

Suggests covr, DT, knitr, optparse, pkgdown, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

Collate dataframe_helpers.R miniTools.R RcheckLibrary.R optparse_helper.R galaxy.R logging.R utils.R

NeedsCompilation no

Author Lain Pavot [aut, cre], Melanie Petera [aut]

Maintainer Lain Pavot <lain.pavot@inrae.fr>

Repository CRAN

Date/Publication 2023-09-08 14:50:02 UTC

R topics documented:

`check_err` *Check Errors*

Description

check_err Generic function stop in error if problems have been encountered

Usage

```
check_err(err_stock)
```

Arguments

`err_stock`	vector of results returned by check functions

Value

`NULL`

Author(s)

M.Petera

`check_one_character` *check_one_character*

Description

Use this function to validate parameters. You're never assured that provided parameters from users are the right type, or length. This may be the case with your own code as well, if you have undetected bugs in your code.

This function helps prevent unpredictable behaviour coming from bad parameters.

It checks the size of vectors, and the type of values. If the parameter is not the good type or length, the program stops with an explanatory error.

Author(s)

L.Pavot

See Also

check_parameter_type,check_parameter_length

check_one_integer,check_one_logical,check_one_numeric

check_one_complex,check_one_character

Examples

```
## here is a simple utility function we will use in this example.
## It is not important
show_last_error <- function(error) {
  dump.frames()
  message(base::attr(last.dump, "error.message"))
}

## The example really starts here
## we have a simple function like this:
custom_message <- function(text) {
  message(sprintf("Message: %s", text))
}

## this function needs to have a character vector as first
## parameter.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(42), error = show_last_error)


## this function needs to have a vector of length 1.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  check_parameter_length(text, 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)


## Or, to be more concise:
custom_message <- function(text) {
  check_param_type_n_length(text, "character", 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)
```

```
tryCatch(custom_message(42), error = show_last_error)


## Let's say the text can be 1 or more elements, and can be null.
custom_message <- function(text) {
  check_param_type_n_length(
    text,
    expected_type = "character",
    or_null = TRUE,
    expected_size = 1,
    or_more = TRUE
  )
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(custom_message(c(42, 43)), error = show_last_error)
tryCatch(custom_message(NULL), error = show_last_error)
## no error, because or_null is TRUE
tryCatch(custom_message(character(0)), error = show_last_error)
tryCatch(custom_message(c("uwu", ":3")), error = show_last_error)
## no error, because or_more is TRUE

## With a function that has a lot of parameters, it may be usefull to
## provide the parameter's number. And, because it becomes very long
## to test all those parameters, we will use shortcuts functions
write_msg <- function(
  text,
  font = "owo",
  font_size = 16,
  italic = FALSE,
  bold = FALSE
) {
  check_one_character(text, nth = "1st")
  check_one_character(font, nth = "2nd")
  check_one_numeric(font_size, nth = "3rd")
  check_one_logical(italic, nth = "before last")
  check_one_logical(bold, nth = "last")
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(write_msg(text = 42, "font", 16), error = show_last_error)
tryCatch(write_msg("uwu", font = 1, 16), error = show_last_error)
tryCatch(write_msg("uwu", font_size = "16"), error = show_last_error)
tryCatch(write_msg("uwu", italic = "FALSE"), error = show_last_error)
tryCatch(write_msg("uwu", bold = "FALSE"), error = show_last_error)
```

---

check_one_complex *check_one_complex*

Description

Use this function to validate parameters. You're never assured that provided parameters from users are the right type, or length. This may be the case with your own code as well, if you have undetected bugs in your code.

This function helps prevent unpredictable behaviour coming from bad parameters.

It checks the size of vectors, and the type of values. If the parameter is not the good type or length, the program stops with an explanatory error.

Author(s)

L.Pavot

See Also

check_parameter_type,check_parameter_length

check_one_integer,check_one_logical,check_one_numeric

check_one_complex,check_one_character

Examples

```
## here is a simple utility function we will use in this example.
## It is not important
show_last_error <- function(error) {
  dump.frames()
  message(base::attr(last.dump, "error.message"))
}

## The example really starts here
## we have a simple function like this:
custom_message <- function(text) {
  message(sprintf("Message: %s", text))
}

## this function needs to have a character vector as first
## parameter.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(42), error = show_last_error)


## this function needs to have a vector of length 1.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  check_parameter_length(text, 1)
  message(base::sprintf("Message: %s", text))
```

```
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)


## Or, to be more concise:
custom_message <- function(text) {
  check_param_type_n_length(text, "character", 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)
tryCatch(custom_message(42), error = show_last_error)


## Let's say the text can be 1 or more elements, and can be null.
custom_message <- function(text) {
  check_param_type_n_length(
    text,
    expected_type = "character",
    or_null = TRUE,
    expected_size = 1,
    or_more = TRUE
  )
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(custom_message(c(42, 43)), error = show_last_error)
tryCatch(custom_message(NULL), error = show_last_error)
## no error, because or_null is TRUE
tryCatch(custom_message(character(0)), error = show_last_error)
tryCatch(custom_message(c("uwu", ":3")), error = show_last_error)
## no error, because or_more is TRUE

## With a function that has a lot of parameters, it may be usefull to
## provide the parameter's number. And, because it becomes very long
## to test all those parameters, we will use shortcuts functions
write_msg <- function(
  text,
  font = "owo",
  font_size = 16,
  italic = FALSE,
  bold = FALSE
) {
  check_one_character(text, nth = "1st")
  check_one_character(font, nth = "2nd")
  check_one_numeric(font_size, nth = "3rd")
  check_one_logical(italic, nth = "before last")
  check_one_logical(bold, nth = "last")
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(write_msg(text = 42, "font", 16), error = show_last_error)
tryCatch(write_msg("uwu", font = 1, 16), error = show_last_error)
tryCatch(write_msg("uwu", font_size = "16"), error = show_last_error)
tryCatch(write_msg("uwu", italic = "FALSE"), error = show_last_error)
tryCatch(write_msg("uwu", bold = "FALSE"), error = show_last_error)
```

check_one_integer *check_one_integer*

Description

Use this function to validate parameters. You're never assured that provided parameters from users are the right type, or length. This may be the case with your own code as well, if you have undetected bugs in your code.

This function helps prevent unpredictable behaviour coming from bad parameters.

It checks the size of vectors, and the type of values. If the parameter is not the good type or length, the program stops with an explanatory error.

Author(s)

L.Pavot

See Also

check_parameter_type,check_parameter_length

check_one_integer,check_one_logical,check_one_numeric

check_one_complex,check_one_character

Examples

```
## here is a simple utility function we will use in this example.
## It is not important
show_last_error <- function(error) {
  dump.frames()
  message(base::attr(last.dump, "error.message"))
}

## The example really starts here
## we have a simple function like this:
custom_message <- function(text) {
  message(sprintf("Message: %s", text))
}

## this function needs to have a character vector as first
## parameter.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(42), error = show_last_error)
```

```
## this function needs to have a vector of length 1.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  check_parameter_length(text, 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)


## Or, to be more concise:
custom_message <- function(text) {
  check_param_type_n_length(text, "character", 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)
tryCatch(custom_message(42), error = show_last_error)


## Let's say the text can be 1 or more elements, and can be null.
custom_message <- function(text) {
  check_param_type_n_length(
    text,
    expected_type = "character",
    or_null = TRUE,
    expected_size = 1,
    or_more = TRUE
  )
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(custom_message(c(42, 43)), error = show_last_error)
tryCatch(custom_message(NULL), error = show_last_error)
## no error, because or_null is TRUE
tryCatch(custom_message(character(0)), error = show_last_error)
tryCatch(custom_message(c("uwu", ":3")), error = show_last_error)
## no error, because or_more is TRUE

## With a function that has a lot of parameters, it may be usefull to
## provide the parameter's number. And, because it becomes very long
## to test all those parameters, we will use shortcuts functions
write_msg <- function(
  text,
  font = "owo",
  font_size = 16,
  italic = FALSE,
  bold = FALSE
) {
  check_one_character(text, nth = "1st")
  check_one_character(font, nth = "2nd")
  check_one_numeric(font_size, nth = "3rd")
  check_one_logical(italic, nth = "before last")
```

```
  check_one_logical(bold, nth = "last")
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(write_msg(text = 42, "font", 16), error = show_last_error)
tryCatch(write_msg("uwu", font = 1, 16), error = show_last_error)
tryCatch(write_msg("uwu", font_size = "16"), error = show_last_error)
tryCatch(write_msg("uwu", italic = "FALSE"), error = show_last_error)
tryCatch(write_msg("uwu", bold = "FALSE"), error = show_last_error)
```

`check_one_logical` *check_one_logical*

Description

Use this function to validate parameters. You're never assured that provided parameters from users are the right type, or length. This may be the case with your own code as well, if you have undetected bugs in your code.

This function helps prevent unpredictable behaviour coming from bad parameters.

It checks the size of vectors, and the type of values. If the parameter is not the good type or length, the program stops with an explanatory error.

Author(s)

L.Pavot

See Also

check_parameter_type,check_parameter_length

check_one_integer,check_one_logical,check_one_numeric

check_one_complex,check_one_character

Examples

```
## here is a simple utility function we will use in this example.
## It is not important
show_last_error <- function(error) {
  dump.frames()
  message(base::attr(last.dump, "error.message"))
}

## The example really starts here
## we have a simple function like this:
custom_message <- function(text) {
  message(sprintf("Message: %s", text))
}
```

```
## this function needs to have a character vector as first
## parameter.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(42), error = show_last_error)


## this function needs to have a vector of length 1.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  check_parameter_length(text, 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)


## Or, to be more concise:
custom_message <- function(text) {
  check_param_type_n_length(text, "character", 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)
tryCatch(custom_message(42), error = show_last_error)


## Let's say the text can be 1 or more elements, and can be null.
custom_message <- function(text) {
  check_param_type_n_length(
    text,
    expected_type = "character",
    or_null = TRUE,
    expected_size = 1,
    or_more = TRUE
  )
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(custom_message(c(42, 43)), error = show_last_error)
tryCatch(custom_message(NULL), error = show_last_error)
## no error, because or_null is TRUE
tryCatch(custom_message(character(0)), error = show_last_error)
tryCatch(custom_message(c("uwu", ":3")), error = show_last_error)
## no error, because or_more is TRUE

## With a function that has a lot of parameters, it may be usefull to
## provide the parameter's number. And, because it becomes very long
## to test all those parameters, we will use shortcuts functions
write_msg <- function(
  text,
  font = "owo",
```

```
  font_size = 16,
  italic = FALSE,
  bold = FALSE
) {
  check_one_character(text, nth = "1st")
  check_one_character(font, nth = "2nd")
  check_one_numeric(font_size, nth = "3rd")
  check_one_logical(italic, nth = "before last")
  check_one_logical(bold, nth = "last")
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(write_msg(text = 42, "font", 16), error = show_last_error)
tryCatch(write_msg("uwu", font = 1, 16), error = show_last_error)
tryCatch(write_msg("uwu", font_size = "16"), error = show_last_error)
tryCatch(write_msg("uwu", italic = "FALSE"), error = show_last_error)
tryCatch(write_msg("uwu", bold = "FALSE"), error = show_last_error)
```

`check_one_numeric` *check_one_numeric*

Description

Use this function to validate parameters. You're never assured that provided parameters from users are the right type, or length. This may be the case with your own code as well, if you have undetected bugs in your code.

This function helps prevent unpredictable behaviour coming from bad parameters.

It checks the size of vectors, and the type of values. If the parameter is not the good type or length, the program stops with an explanatory error.

Author(s)

L.Pavot

See Also

check_parameter_type,check_parameter_length

check_one_integer,check_one_logical,check_one_numeric

check_one_complex,check_one_character

Examples

```
## here is a simple utility function we will use in this example.
## It is not important
show_last_error <- function(error) {
  dump.frames()
  message(base::attr(last.dump, "error.message"))
```

```
}

## The example really starts here
## we have a simple function like this:
custom_message <- function(text) {
  message(sprintf("Message: %s", text))
}

## this function needs to have a character vector as first
## parameter.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(42), error = show_last_error)


## this function needs to have a vector of length 1.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  check_parameter_length(text, 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)


## Or, to be more concise:
custom_message <- function(text) {
  check_param_type_n_length(text, "character", 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)
tryCatch(custom_message(42), error = show_last_error)


## Let's say the text can be 1 or more elements, and can be null.
custom_message <- function(text) {
  check_param_type_n_length(
    text,
    expected_type = "character",
    or_null = TRUE,
    expected_size = 1,
    or_more = TRUE
  )
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(custom_message(c(42, 43)), error = show_last_error)
tryCatch(custom_message(NULL), error = show_last_error)
## no error, because or_null is TRUE
tryCatch(custom_message(character(0)), error = show_last_error)
tryCatch(custom_message(c("uwu", ":3")), error = show_last_error)
```

```
## no error, because or_more is TRUE

## With a function that has a lot of parameters, it may be usefull to
## provide the parameter's number. And, because it becomes very long
## to test all those parameters, we will use shortcuts functions
write_msg <- function(
  text,
  font = "owo",
  font_size = 16,
  italic = FALSE,
  bold = FALSE
) {
  check_one_character(text, nth = "1st")
  check_one_character(font, nth = "2nd")
  check_one_numeric(font_size, nth = "3rd")
  check_one_logical(italic, nth = "before last")
  check_one_logical(bold, nth = "last")
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(write_msg(text = 42, "font", 16), error = show_last_error)
tryCatch(write_msg("uwu", font = 1, 16), error = show_last_error)
tryCatch(write_msg("uwu", font_size = "16"), error = show_last_error)
tryCatch(write_msg("uwu", italic = "FALSE"), error = show_last_error)
tryCatch(write_msg("uwu", bold = "FALSE"), error = show_last_error)
```

`check_parameter_length` *check_parameter_length - validate parameter's length*

Description

Use this function to validate parameters. You're never assured that provided parameters from users are the right type, or length. This may be the case with your own code as well, if you have undetected bugs in your code.

This function helps prevent unpredictable behaviour coming from bad parameters.

It checks the size of vectors, and the type of values. If the parameter is not the good type or length, the program stops with an explanatory error.

Usage

```
check_parameter_length(
  value,
  expected_size,
  nth = NULL,
  func_name = NULL,
  param_name = NULL,
  or_more = FALSE,
  nframe = 1
)
```

Arguments

value	The parameter to test.
expected_size	The expected size of the vector. Usualy, 1.
nth	This parameter is used in the error message generation. Provide a character vector like "first", "second", "1st", "2nd", ... this must be the number of the parameter if the function.
func_name	By default, the function name is guessed from the stack. But if you want to change it, or if it is not the right function name in error messages, set the right one here.
param_name	Like func_name, by default the param name is guessed. But if you want to change it, or if it is not the right parameter name in error messages, set the right one here.
or_more	When we check the parameter's length, if or_more is TRUE and the value is bigger than expected_size, then, the length check does not occur
nframe	The number of function calls between this function and the function where the value to test is a parameter. for example, if a user calls function A, which calls check_param_* directly, then nframe must be 1 because it is a direct call. But, if the user has called function A, and function A calls function B, and check_param_*, is called in function B, then, for check_param_* to understant it is a parameter comming from function A (and not from function B), we have to tell check_param_* that nframe is 2. If the function name is not the right name, it may be because of that. So don't fear testing different values for nframes.

Author(s)

L.Pavot

See Also

check_parameter_type,check_parameter_length

check_one_integer,check_one_logical,check_one_numeric

check_one_complex,check_one_character

Examples

```
## here is a simple utility function we will use in this example.
## It is not important
show_last_error <- function(error) {
  dump.frames()
  message(base::attr(last.dump, "error.message"))
}

## The example really starts here
## we have a simple function like this:
custom_message <- function(text) {
  message(sprintf("Message: %s", text))
```

```
}

## this function needs to have a character vector as first
## parameter.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(42), error = show_last_error)


## this function needs to have a vector of length 1.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  check_parameter_length(text, 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)


## Or, to be more concise:
custom_message <- function(text) {
  check_param_type_n_length(text, "character", 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)
tryCatch(custom_message(42), error = show_last_error)


## Let's say the text can be 1 or more elements, and can be null.
custom_message <- function(text) {
  check_param_type_n_length(
    text,
    expected_type = "character",
    or_null = TRUE,
    expected_size = 1,
    or_more = TRUE
  )
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(custom_message(c(42, 43)), error = show_last_error)
tryCatch(custom_message(NULL), error = show_last_error)
## no error, because or_null is TRUE
tryCatch(custom_message(character(0)), error = show_last_error)
tryCatch(custom_message(c("uwu", ":3")), error = show_last_error)
## no error, because or_more is TRUE

## With a function that has a lot of parameters, it may be usefull to
## provide the parameter's number. And, because it becomes very long
## to test all those parameters, we will use shortcuts functions
write_msg <- function(
```

```
  text,
  font = "owo",
  font_size = 16,
  italic = FALSE,
  bold = FALSE
) {
  check_one_character(text, nth = "1st")
  check_one_character(font, nth = "2nd")
  check_one_numeric(font_size, nth = "3rd")
  check_one_logical(italic, nth = "before last")
  check_one_logical(bold, nth = "last")
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(write_msg(text = 42, "font", 16), error = show_last_error)
tryCatch(write_msg("uwu", font = 1, 16), error = show_last_error)
tryCatch(write_msg("uwu", font_size = "16"), error = show_last_error)
tryCatch(write_msg("uwu", italic = "FALSE"), error = show_last_error)
tryCatch(write_msg("uwu", bold = "FALSE"), error = show_last_error)
```

`check_parameter_type` *check_parameter_type - validate parameter's type*

Description

Use this function to validate parameters. You're never assured that provided parameters from users are the right type, or length. This may be the case with your own code as well, if you have undetected bugs in your code.

This function helps prevent unpredictable behaviour coming from bad parameters.

It checks the size of vectors, and the type of values. If the parameter is not the good type or length, the program stops with an explanatory error.

Usage

```
check_parameter_type(
  value,
  expected_type,
  nth = NULL,
  func_name = NULL,
  param_name = NULL,
  or_null = FALSE,
  nframe = 1
)
```

Arguments

`value`	The parameter to test.
`expected_type`	The `chararcter` vector of the kind: `"character"`, `"integer"`, `"logical"`, ...

nth	This parameter is used in the error message generation. Provide a character vector like "first", "second", "1st", "2nd", ... this must be the number of the parameter if the function.
func_name	By default, the function name is guessed from the stack. But if you want to change it, or if it is not the right function name in error messages, set the right one here.
param_name	Like func_name, by default the param name is guessed. But if you want to change it, or if it is not the right parameter name in error messages, set the right one here.
or_null	When we check the parameter's type, if or_null is TRUE and the value is NULL, then, the type check does not occur
nframe	The number of function calls between this function and the function where the value to test is a parameter. for example, if a user calls function A, which calls check_param_* directly, then nframe must be 1 because it is a direct call. But, if the user has called function A, and function A calls function B, and check_param_*, is called in function B, then, for check_param_* to understant it is a parameter comming from function A (and not from function B), we have to tell check_param_* that nframe is 2. If the function name is not the right name, it may be because of that. So don't fear testing different values for nframes.

Author(s)

L.Pavot

See Also

check_parameter_type,check_parameter_length

check_one_integer,check_one_logical,check_one_numeric

check_one_complex,check_one_character

Examples

```
## here is a simple utility function we will use in this example.
## It is not important
show_last_error <- function(error) {
  dump.frames()
  message(base::attr(last.dump, "error.message"))
}

## The example really starts here
## we have a simple function like this:
custom_message <- function(text) {
  message(sprintf("Message: %s", text))
}

## this function needs to have a character vector as first
## parameter.
## So, to validate the parameter, we could write:
```

```
custom_message <- function(text) {
  check_parameter_type(text, "character")
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(42), error = show_last_error)


## this function needs to have a vector of length 1.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  check_parameter_length(text, 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)


## Or, to be more concise:
custom_message <- function(text) {
  check_param_type_n_length(text, "character", 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)
tryCatch(custom_message(42), error = show_last_error)


## Let's say the text can be 1 or more elements, and can be null.
custom_message <- function(text) {
  check_param_type_n_length(
    text,
    expected_type = "character",
    or_null = TRUE,
    expected_size = 1,
    or_more = TRUE
  )
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(custom_message(c(42, 43)), error = show_last_error)
tryCatch(custom_message(NULL), error = show_last_error)
## no error, because or_null is TRUE
tryCatch(custom_message(character(0)), error = show_last_error)
tryCatch(custom_message(c("uwu", ":3")), error = show_last_error)
## no error, because or_more is TRUE

## With a function that has a lot of parameters, it may be usefull to
## provide the parameter's number. And, because it becomes very long
## to test all those parameters, we will use shortcuts functions
write_msg <- function(
  text,
  font = "owo",
  font_size = 16,
  italic = FALSE,
  bold = FALSE
```

```
) {
  check_one_character(text, nth = "1st")
  check_one_character(font, nth = "2nd")
  check_one_numeric(font_size, nth = "3rd")
  check_one_logical(italic, nth = "before last")
  check_one_logical(bold, nth = "last")
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(write_msg(text = 42, "font", 16), error = show_last_error)
tryCatch(write_msg("uwu", font = 1, 16), error = show_last_error)
tryCatch(write_msg("uwu", font_size = "16"), error = show_last_error)
tryCatch(write_msg("uwu", italic = "FALSE"), error = show_last_error)
tryCatch(write_msg("uwu", bold = "FALSE"), error = show_last_error)
```

`check_param_type_n_length`

check_param_type_n_length - to check parameters

Description

Use this function to validate parameters. You're never assured that provided parameters from users are the right type, or length. This may be the case with your own code as well, if you have undetected bugs in your code.

This function helps prevent unpredictable behaviour coming from bad parameters.

It checks the size of vectors, and the type of values. If the parameter is not the good type or length, the program stops with an explanatory error.

Usage

```
check_param_type_n_length(
  value,
  expected_type,
  expected_size = 1,
  nth = NULL,
  func_name = NULL,
  param_name = NULL,
  or_more = FALSE,
  or_null = FALSE,
  nframe = 1
)
```

Arguments

`value`	The parameter to test.
`expected_type`	The `chararcter` vector of the kind: `"character"`, `"integer"`, `"logical"`, ...
`expected_size`	The expected `size` of the vector. Usualy, `1`.

nth	This parameter is used in the error message generation. Provide a character vector like "first", "second", "1st", "2nd", ... this must be the number of the parameter if the function.
func_name	By default, the function name is guessed from the stack. But if you want to change it, or if it is not the right function name in error messages, set the right one here.
param_name	Like func_name, by default the param name is guessed. But if you want to change it, or if it is not the right parameter name in error messages, set the right one here.
or_more	When we check the parameter's length, if or_more is TRUE and the value is bigger than expected_size, then, the length check does not occur
or_null	When we check the parameter's type, if or_null is TRUE and the value is NULL, then, the type check does not occur
nframe	The number of function calls between this function and the function where the value to test is a parameter. for example, if a user calls function A, which calls check_param_* directly, then nframe must be 1 because it is a direct call. But, if the user has called function A, and function A calls function B, and check_param_*, is called in function B, then, for check_param_* to understant it is a parameter comming from function A (and not from function B), we have to tell check_param_* that nframe is 2. If the function name is not the right name, it may be because of that. So don't fear testing different values for nframes.

Author(s)

L.Pavot

See Also

check_parameter_type,check_parameter_length

check_one_integer,check_one_logical,check_one_numeric

check_one_complex,check_one_character

Examples

```
## here is a simple utility function we will use in this example.
## It is not important
show_last_error <- function(error) {
  dump.frames()
  message(base::attr(last.dump, "error.message"))
}

## The example really starts here
## we have a simple function like this:
custom_message <- function(text) {
  message(sprintf("Message: %s", text))
}
```

```
## this function needs to have a character vector as first
## parameter.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(42), error = show_last_error)


## this function needs to have a vector of length 1.
## So, to validate the parameter, we could write:
custom_message <- function(text) {
  check_parameter_type(text, "character")
  check_parameter_length(text, 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)


## Or, to be more concise:
custom_message <- function(text) {
  check_param_type_n_length(text, "character", 1)
  message(base::sprintf("Message: %s", text))
}
tryCatch(custom_message(c("uwu", "owo")), error = show_last_error)
tryCatch(custom_message(42), error = show_last_error)


## Let's say the text can be 1 or more elements, and can be null.
custom_message <- function(text) {
  check_param_type_n_length(
    text,
    expected_type = "character",
    or_null = TRUE,
    expected_size = 1,
    or_more = TRUE
  )
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(custom_message(c(42, 43)), error = show_last_error)
tryCatch(custom_message(NULL), error = show_last_error)
## no error, because or_null is TRUE
tryCatch(custom_message(character(0)), error = show_last_error)
tryCatch(custom_message(c("uwu", ":3")), error = show_last_error)
## no error, because or_more is TRUE

## With a function that has a lot of parameters, it may be usefull to
## provide the parameter's number. And, because it becomes very long
## to test all those parameters, we will use shortcuts functions
write_msg <- function(
  text,
  font = "owo",
```

```
  font_size = 16,
  italic = FALSE,
  bold = FALSE
) {
  check_one_character(text, nth = "1st")
  check_one_character(font, nth = "2nd")
  check_one_numeric(font_size, nth = "3rd")
  check_one_logical(italic, nth = "before last")
  check_one_logical(bold, nth = "last")
  message(paste0(base::sprintf("Message: %s", text), collapse = "\n"))
}
tryCatch(write_msg(text = 42, "font", 16), error = show_last_error)
tryCatch(write_msg("uwu", font = 1, 16), error = show_last_error)
tryCatch(write_msg("uwu", font_size = "16"), error = show_last_error)
tryCatch(write_msg("uwu", italic = "FALSE"), error = show_last_error)
tryCatch(write_msg("uwu", bold = "FALSE"), error = show_last_error)
```

`collapse` *collapse - to paste strings with collapse = ""*

Description

collapse does exactly when paste does, but default collapse = ""

Usage

```
collapse(..., sep = "")
```

Arguments

`...`	passed to `base::paste0()`
`sep`	set the separator. Deafult is ""

Author(s)

L.Pavot

Examples

```
collapse(list("a message ", "in multiple", "parts"))
```

collapse_lines *collapse_lines - to paste strings with collapse = "\n"*

Description

collapse_lines() does exactly when paste does, but default collapse = "\n"

Usage

```
collapse_lines(..., sep = "\n")
```

Arguments

...	passed to base::paste0()
sep	set the separator. Deafult is "\n"

Author(s)

L.Pavot

Examples

```
collapse_lines(list("a message ", "in multiple", "parts"))
```

convert_parameters *Convert Parameters*

Description

convert_parameters Applies a list of converters to each values on a list. If a value is modified during the conversion (successfull conversion) then, no further convert will be applied to this value, so values are only converted once.

Usage

```
convert_parameters(args, converters)
```

Arguments

args	a named list, which values will be converted.
converters	a vector of function. Each function will be applied to each values with the exception of values already converted by a previous converter.

Value

a named `list` object with values converted by converters.

Author(s)

L.Pavot

Examples

```
boolean_converter <- function(x) {
  return(if (x == "TRUE") TRUE else if (x == "FALSE") FALSE else x)
}
parameters <- W4MRUtils::convert_parameters(
  list("x" = "TRUE"),
  c(boolean_converter)
)
print(parameters$`some-parameter`)
## "TRUE" has becomes TRUE.
```

---

df_force_numeric *Convert data frame to numeric.*

Description

df_force_numeric Converts integer columns of a data frame into numeric.

Usage

```
df_force_numeric(df, cols = NULL)
```

Arguments

`df`	The data frame.
`cols`	The set of columns to convert to numeric. By default (when set to `NULL`) all integer columns are converted. Set it to a character vector containing the names of the columns you want to convert, or ton integer vector containing the indices of the columns. Can be used to force conversion of non integer columns.

Value

The converted `data.frame`.

Examples

```
# Convert an integer data frame
df <- data.frame(a = as.integer(c(1, 4)), b = as.integer(c(6, 5)))
df <- W4MRUtils::df_force_numeric(df)
```

df_is *Test type of a data frame.*

Description

df_is This function tests if the columns of a data frame are all of the same type.

Usage

```
df_is(df, type)
```

Arguments

df	The data frame.
type	The type you expect the columns to have. It must be one of the R base types: - 'character' ; - 'factor' ; - 'integer' ; - 'numeric' ; - 'logical'.

Value

TRUE or FALSE.

Examples

```
# Test if a data frame contains only integers
df <- data.frame(a = c(1, 4), b = c(6, 5))
# should return FALSE since in R all integers are converted to
# numeric by default.
W4MRUtils::df_is(df, "integer")
# should return TRUE.
W4MRUtils::df_is(df, "numeric")
```

df_read_table *Data frame loading from a file.*

Description

df_read_table Reads a data frame from a file and possibly convert integer columns to numeric. This function calls the built-in `read.table()` method and then `W4MRUtils::df_force_numeric()`.

Usage

```
df_read_table(file, force_numeric = FALSE, ...)
```

Arguments

file	The path to the file you want to load. See read.table() documentation for more information.
force_numeric	If set to TRUE, all integer columns will be converted to numeric.
...	Parameter to transmit to the read.table function.

Value

The loaded data frame.

Examples

```
# Load a data frame from a file and convert integer columns
file_path <- system.file(
  "extdata",
  "example_df_read_table.csv",
  package="W4MRUtils"
)
str(W4MRUtils::df_read_table(
  file_path,
  sep = ",",
  force_numeric = TRUE,
  header=TRUE
))
```

---

get_base_dir *get_base_dir - to get... the base directory*

Description

get_base_dir

Usage

```
get_base_dir()
```

Value

the directory path of the main script. PWD otherwise.

Examples

```
print(get_base_dir())
```

`get_logger` *Instantiate a Logger*

Description

Create a logger of the given name. You can call again `get_logger` and provide the same name to get the same logger. It will not be recreated unless recreate is TRUE.

Usage

```
get_logger(name, recreate = FALSE, ...)
```

Arguments

`name`	the name of the logger to get or create. This name will be used in logs to differentiate from which part of you program comes which lines of log. See the example of usage bellow.
`recreate`	logical=FALSE tells whether to recreate the logger of the given name or not. Preferably, one should not recreate a new logger each time.
`...`	Arguments passed on to `W4MLogger`

Details

get_logger

Value

A new W4MLogger instance if it did not exist or if recreate is TRUE. Otherwise, a new W4MLogger instance.

Author(s)

L.Pavot

Examples

```
## let's say our program is divided in three big tasks:
##   - the parsing
##   - the processing
##   - the output writing
parser_logger <- W4MRUtils::get_logger("Parser")
process_logger <- W4MRUtils::get_logger("Processing")
write_logger <- W4MRUtils::get_logger("Writer")
input_path <- "/home/anyone/input.csv"
parser_logger$info(paste("Parsing the input file at", input_path))
parser_logger$debug("Input extension detected: csv")
parser_logger$debug("The csv parser program will be used")
```

```
## do the parsing...
input <- list(a=1:5, b=5:10, c=8:2)
parser_logger$info("Parsing succeed")
process_logger$info("Starting the processing of:", input)
process_logger$debug("The processing has started at...")
result <- as.list(input)
process_logger$debug("The processing has finished at...")
process_logger$info("Processing finished in x seconds.")
outfile <- "/home/anyone/output.tsv"
write_logger$info(paste("Creating the output in", outfile))

## we detected that the input was csv and the out was tsv:
## but it is not a blocking problem
write_logger$warning("The input and output file's extensions are different")
write_logger$debug("The output will be casted from csv to tsv")

## we try to write the file, but it fails
tryCatch({
  ## writing the output file failed with this error:
  stop(sprintf("I/O Error: %s is not writable.", outfile))
}, error = function(e) {
  write_logger$error(e$message)
  write_logger$error("Writing output file aborted.")
  ## quit(save = "no", status = 42)
})

## note that debug output were not written. To show debug logs
## we have to active it (disabled by default):

write_logger$set_debug()
write_logger$debug("The debug outputs are now visible!")
```

`get_r_env` *get_r_env - provides env vars begining with R_**

Description

Returns a list of env vars if the start with R_*.

Usage

```
get_r_env()
```

Value

a `list` of environment variables which begin by `R_`.

Author(s)

L.Pavot

See Also

run_galaxy_processing

`import2` *Import two W4M tables*

Description

import2 Function to import a metadata table file and its corresponding dataMatrix file. import2 performs checks to ensure the identifiers match between the two tables and stops with an explicit error message in case identifiers do not match.

Usage

```
import2(pathDM, pathMeta, typeMeta, disable_comm = TRUE)
```

Arguments

`pathDM`	a path to a file corresponding to the dataMatrix
`pathMeta`	a path to a file corresponding to the metadata table
`typeMeta`	"sample" or "variable" depending on the metadata content
`disable_comm`	a `boolean` with default to `TRUE` to indicate whether the comment character # should be disabled as a comment tag for the import of the metadata file; when `TRUE`, # in the metadata table's columns will be considered as any other character.

Value

a `list` containing two elements:

- dataMatrix a `data.frame` corresponding to the imported dataMatrix table;
- metadata a `data.frame` corresponding to the imported metadata table

Author(s)

M.Petera

Examples

```
dm_path <- system.file(
  "extdata",
  "mini_datamatrix.txt",
  package="W4MRUtils"
)
meta_path <- system.file(
```

```
  "extdata",
  "mini_variablemetadata.txt",
  package="W4MRUtils"
)

## import considering # is not a comment character
A <- W4MRUtils::import2(dm_path, meta_path, "variable")
print(A$dataMatrix[1:5, 1:5])
print(A$metadata[1:5, ])

## import considering # is a comment character
B <- W4MRUtils::import2(dm_path, meta_path, "variable", disable_comm = FALSE)
print(B$dataMatrix[1:5, 1:5])
print(B$metadata[1:5, ])
```

`import3` *Import the three W4M tables*

Description

import3 Function to import the three W4M tables from files (dataMatrix, sampleMetadata, variableMetadata) import3 performs checks to ensure the identifiers match between the three tables and stops with an explicit error message in case identifiers do not match.

Usage

```
import3(pathDM, pathSM, pathVM, disable_comm = TRUE)
```

Arguments

`pathDM`	a path to a file corresponding to the dataMatrix
`pathSM`	a path to a file corresponding to the sampleMetadata
`pathVM`	a path to a file corresponding to the variableMetadata
`disable_comm`	a `boolean` with default to `TRUE` to indicate whether the comment character # should be disabled as a comment tag for the import of the metadata files; when `TRUE`, # in the metadata table's columns will be considered as any other character.

Value

a `list` containing three elements:

- dataMatrix a `data.frame` corresponding to the imported dataMatrix table;
- sampleMetadata a `data.frame` corresponding to the imported sampleMetadata table;
- variableMetadata a `data.frame` corresponding to the imported variableMetadata table

Author(s)

M.Petera

Examples

```
dm_path <- system.file(
  "extdata",
  "mini_datamatrix.txt",
  package="W4MRUtils"
)
vm_path <- system.file(
  "extdata",
  "mini_variablemetadata.txt",
  package="W4MRUtils"
)
sm_path <- system.file(
  "extdata",
  "mini_samplemetadata.txt",
  package="W4MRUtils"
)

## import considering # is not a comment character
A <- W4MRUtils::import3(dm_path, sm_path, vm_path)
print(A$dataMatrix[1:5, 1:5])
print(A$sampleMetadata[1:5, ])
print(A$variableMetadata[1:5, ])

## import considering # is a comment character
B <- W4MRUtils::import3(dm_path, sm_path, vm_path, disable_comm = FALSE)
print(B$dataMatrix[1:5, 1:5])
print(B$sampleMetadata[1:5, ])
print(B$variableMetadata[1:5, ])
```

---

in_galaxy_env *in_galaxy_env - check if the script has been run by galaxy*

Description

in_galaxy_env returns TRUE if it detects some galaxy-specific environment variables. FALSE otherwise.

Usage

```
in_galaxy_env()
```

Value

A logical - whether the script has been run by galaxy or not.

`match2` *Table match check functions*

Description

match2 To check if data_matrix and (variable or sample)metadata match regarding identifiers

Usage

```
match2(data_matrix, metadata, metadata_type)
```

Arguments

`data_matrix`	data.frame containing data_matrix
`metadata`	data.frame containing sample_metadata or variable_metadata
`metadata_type`	"sample" or "variable" depending on metadata content

Value

`character` vector a list of errors encountered

Author(s)

M.Petera

`match3` *match3*

Description

match3 To check if the 3 standard tables match regarding identifiers

Usage

```
match3(data_matrix, sample_metadata, variable_metadata)
```

Arguments

`data_matrix`	data.frame containing data_matrix
`sample_metadata`	data.frame containing sample_metadata
`variable_metadata`	data.frame containing variable_metadata

Value

`character` vector a list of errors encountered

Author(s)

M.Petera

`mini_tools` *Mini tools for Galaxy scripting*

Description

Mini tools for Galaxy scripting Mini tools for Galaxy scripting Coded by: M.Petera,

R functions to use in R scripts and wrappers to make things easier (lightening code, reducing verbose...)

V0: script structure + first functions V1: addition of functions to handle special characters in identifiers

---

`optparse_character` *optparse_character - define a command parameter as string*

Description

To be used with `optparse_parameters`. This function tells the provided parameter is to be parsed as a single string.

Usage

```
optparse_character(help = "No documentation yet.", short = NULL, default = 0)
```

Arguments

`help`	• The help string to display when –help is triggered
`short`	• The shortcut fir this parameter. For example for a –output param, we could use `optparse_flag(short = "-o", ...)` to set the "-o" shortcut.
`default`	• The default value this parameter will hold.

Author(s)

L.Pavot

See Also

`optparse_parameters()`

Examples

```
str(optparse_parameters(
  a_parameter = optparse_character(),
  args = list("--a-parameter", "42")
))
```

---

optparse_flag *optparse_flag - define a command parameter as a trigger*

Description

To be used with optparse_parameters. This function tells the provided parameter is a trigger (logical - TRUE/FALSE). When the trigger parameter is not provided in the command line, the value is FALSE. Otherwise, it is TRUE.

Usage

```
optparse_flag(help = "No documentation yet.", short = NULL, default = FALSE)
```

Arguments

help	• The help string to display when –help is triggered
short	• The shortcut fir this parameter. For example for a –output param, we could use optparse_flag(short = "-o", ...) to set the "-o" shortcut.
default	• The default value this parameter will hold.

Value

a list to give to optparse_parameters to build the whole command line parsing tool.

Author(s)

L.Pavot

See Also

optparse_parameters()

Examples

```
str(optparse_parameters(
  a_parameter = optparse_flag(),
  args = list("--a-parameter")
))
```

optparse_integer *optparse_integer - define a command parameter as an integer*

Description

To be used with optparse_parameters. This function tells the provided parameter is to be parsed as an integer.

Usage

```
optparse_integer(help = "No documentation yet.", short = NULL, default = 0)
```

Arguments

help	• The help string to display when –help is triggered
short	• The shortcut fir this parameter. For example for a –output param, we could use optparse_flag(short = "-o", ...) to set the "-o" shortcut.
default	• The default value this parameter will hold.

Author(s)

L.Pavot

See Also

optparse_parameters()

Examples

```
str(optparse_parameters(
  a_parameter = optparse_integer(),
  args = list("--a-parameter", "42")
))
```

optparse_list *optparse_list - define a command parameter as a list of objects*

Description

To be used with optparse_parameters. This function tells the provided parameter is to be parsed as a list of objects. The of parameter tells what type are elements of the list. Each element must be separated by a separator. This separator must be the value given in the sep parameter

Usage

```
optparse_list(
  help = "No documentation yet.",
  short = NULL,
  default = "",
  of = "character",
  sep = ",",
  truevalues = c("TRUE", "true", "1", "t", "T")
)
```

Arguments

help	• The help string to display when –help is triggered
short	• The shortcut fir this parameter. For example for a –output param, we could use optparse_flag(short = "-o", ...) to set the "-o" shortcut.
default	• The default value this parameter will hold.
of	• This type of elements of this list
sep	• This character to split on, to get the list
truevalues	• A character vector of different string values to translate it as TRUE value.

Author(s)

L.Pavot

See Also

optparse_parameters()

Examples

```
str(optparse_parameters(
  a_parameter = optparse_list(of="numeric"),
  b_parameter = optparse_list(of="integer"),
  c_parameter = optparse_list(of="logical"),
  args = list(
    "--a-parameter", "42.7,72.5",
    "--b-parameter", "42.7,72.5",
    "--c-parameter", "TRUE,FALSE,FALSE,TRUE"
  )
))
```

`optparse_numeric` *optparse_numeric - define a command parameter as an numeric*

Description

To be used with `optparse_parameters`. This function tells the provided parameter is to be parsed as an numeric.

Usage

```
optparse_numeric(help = "No documentation yet.", short = NULL, default = 0)
```

Arguments

`help`	• The help string to display when –help is triggered
`short`	• The shortcut fir this parameter. For example for a –output param, we could use `optparse_flag(short = "-o", ...)` to set the "-o" shortcut.
`default`	• The default value this parameter will hold.

Author(s)

L.Pavot

See Also

`optparse_parameters()`

Examples

```
str(optparse_parameters(
  a_parameter = optparse_numeric(),
  args = list("--a-parameter", "42.72")
))
```

`optparse_parameters` *optparse_parameters - parse easily the command line parameters*

Description

This function is made to be used with the functions optparse_flag, optparse_numeric, optparse_integer, optparse_character and/or optparse_list

`optparse_parameters` parses arguments based on its parameters.

You just have to call `optparse_parameters` with named arguments. Each parameter is the result of either optparse_flag, optparse_numeric, optparse_integer, optparse_character or optparse_list

Usage

```
optparse_parameters(
  fix_hyphens = TRUE,
  fix_dots = TRUE,
  add_trailing_hyphens = TRUE,
  args = NULL,
  no_optparse = FALSE,
  ...
)
```

Arguments

fix_hyphens	logical - whether to turn underscores into hyphens or not
fix_dots	logical - whether to turn points into hyphens or not
add_trailing_hyphens	logical - whether to add trailing hyphens if missing
args	list - The parameters from the commandArgs function
no_optparse	logical - INTERNAL Tells whether to use optparse library or not
...	parameters definition. Must be the result of either those functions: • optparse_flag • optparse_numeric • optparse_integer • optparse_character • optparse_list

Author(s)

L.Pavot

Examples

```
args <- optparse_parameters(
  a_integer = optparse_integer(),
  a_float = optparse_numeric(),
  a_boolean = optparse_flag(),
  a_character = optparse_character(),
  a_list = optparse_list(of = "numeric"),
  a_char_list = optparse_list(of = "character"),
  a_int_list = optparse_list(of = "integer"),
  args = list(
    "--a-integer",
    "42",
    "--a-float",
    "3.14",
    "--a-boolean",
    "FALSE",
    "--a-character",
```

```
    "FALSE",
    "--a-list",
    "1.5,2,3",
    "--a-char-list",
    "1.5,2,3",
    "--a-int-list",
    "1.5,2,3"
  )
)

str(args)
```

`parse_args` *Parse Command arguments*

Description

parse_args Replacement for the parseCommandArgs utility from batch. Note that inputs like `script.R some-list c(1, 2, 3` will result in args$`some-list` to be the string "c(1, 2, 3)", and not a vector anymore as this ability was permitted by dangerous behaviours from the batch package (the usage of `eval` which MUST NEVER be used on user's inputs).

To get a list of numeric from users, instead of using the `c(1, 2)` trick, please, use regular lists parsing:

```
> args$`some-list`
[1] "1,2"
args$`some-list` <- as.numeric(strsplit(args$`some-list`, ",")[[1]])
> args$`some-list`
[1] 1 2
```

Usage

```
parse_args(
  args = NULL,
  convert_booleans = TRUE,
  convert_numerics = TRUE,
  strip_trailing_dash = TRUE,
  replace_dashes = TRUE
)
```

Arguments

`args`	optional, provide arguments to parse. This function will use 'commandArgs()' if args is not provided
`convert_booleans`	logical - tells the function to convert values into logical if their value is "TRUE" or "FALSE".

convert_numerics
: logical - tells the function to convert values into numeric if possible.

strip_trailing_dash
: • tells whether to remove trailing hyphens from the start of the parameter name

replace_dashes
: • tells whether to turn trailing hyphens into underscores

Value

a named `list` object containing the input parameters in values and the parameters names in names

Author(s)

L.Pavot

Examples

```
## faking command line parameters:

commandArgs <- function() {
  list(
    "--args",
    "param1", "a value",
    "param2", "42"
  )
}

## extracting command line parameters:
parameters <- W4MRUtils::parse_args(args = commandArgs())
str(parameters)
```

---

printf *printf - to format a string and print it*

Description

printf calls sprintf of its parameters to build the error message and prints with the given message

Usage

```
printf(...)
```

Arguments

...
: Arguments passed on to `base::sprintf`

`fmt` a character vector of format strings, each of up to 8192 bytes.

Author(s)

L.Pavot

Examples

```
file <- "/tmp/test"
printf("Error in file: ", file)
```

---

printfp *printfp - to paste, format and print a string*

Description

printfp calls paste and sprintf of its parameters to build the error message and prints with the given message

Usage

```
printfp(x, ...)
```

Arguments

x	a list of format string to concatenate before using sprintf on it.
...	Arguments passed on to base::paste sep a character string to separate the terms. Not NA_character_. collapse an optional character string to separate the results. Not NA_character_. recycle0 logical indicating if zero-length character arguments should lead to the zero-length character(0) after the sep-phase (which turns into "" in the collapse-phase, i.e., when collapse is not NULL).

Author(s)

L.Pavot

Examples

```
file <- "/tmp/test"
printfp(
  list(
    "Very log error message that needs to be cut on multiple lines,",
    "and paste back together, but there are formatings like",
    "%%s for example, that provides a placeholder for parameters.",
    "Here %%s value is %s."
  ), file
)
```

printp *printp - to format a string and print it*

Description

printp calls sprintf of its parameters to build the error message and prints with the given message

Usage

```
printp(...)
```

Arguments

...	Arguments passed on to base::sprintf
	fmt a character vector of format strings, each of up to 8192 bytes.

Author(s)

L.Pavot

Examples

```
file <- "/tmp/test"
printp("Error in file: ", file)
```

reproduce_id *Reproduce ID*

Description

reproduce_id reproduce_id() reinjects original identifiers and original order into final tables

Usage

```
reproduce_id(data_matrix, metadata, metadata_type, id_match)
```

Arguments

data_matrix	data.frame containing data_matrix
metadata	data.frame containing samplemetadata or variablemetadata
metadata_type	"sample" or "variable" depending on metadata content
id_match	'id_match' element produced by stock_id

Value

a named list with two elements: data_matrix: the processed data matrix with its original names and order metadata: the processed metadata, with its original names and order.

Author(s)

M.Petera

Examples

```
myDM <- data.frame(data = 1:6, a = 2:7, b = 3:8, c = 2:7, d = 3:8, e = 2:7)
myvM <- data.frame(variable = 1:6, x = 4:9, y = 2:7, z = 3:8)

A <- W4MRUtils::stock_id(myDM, myvM, "variable")
myDM <- A$dataMatrix
myvM <- A$Metadata
A <- A$id.match

## processing that may change order or requires specific identifiers format
## ...
datamatrix <- as.data.frame(myDM)
sample_metadata <- as.data.frame(myvM)

B <- W4MRUtils::reproduce_id(datamatrix, sample_metadata, "variable", A)
datamatrix <- B$dataMatrix
sample_metadata <- B$Metadata
```

---

run_galaxy_function *run_galaxy_function - automate running functions in galaxy*

Description

This function executes the provided function as a galaxy processing This provided function is expected to take two parameters:

- args, a list of command line parameters
- logger, the logger created for the tool

Usage

```
run_galaxy_function(tool_name, func, ...)
```

Arguments

`tool_name`	• The name of the tool
`func`	• The function to be run, after galaxy header is displayed
`...`	• Parameters propagated to run_galaxy_processing

Author(s)

L.Pavot

See Also

run_galaxy_processing

`run_galaxy_processing` *run_galaxy_processing - automate running code in galaxy*

Description

run_galaxy_processing takes the tool's name, and the code to execute. It detects galaxy-specific environement variable, and show headers and footer if we are in a galaxy env.

It will automatically convert command line parameters using W4MRUtils::parse_args if args is not provided.

Then, it unmangles galaxy parameters (galaxy params / values can be mangled if they contains special characters)

It creates a logger, and provide access to the `logger` and `args` variables from withing the code to execute.

Also, before executing the code, if `source_files` is set to some paths, these paths will be source'd, so the code has access to functions defined in these scripts.

Usage

```
run_galaxy_processing(
  tool_name,
  code,
  tool_version = "unknown",
  unmangle_parameters = TRUE,
  args = NULL,
  logger = NULL,
  source_files = c(),
  env = NULL,
  do_traceback = FALSE
)
```

Arguments

tool_name	• Mandatory. The name of the tool to run.
code	• Mandatory. The code to run the tool
tool_version	• The version number of the tool to run.
unmangle_parameters	• Whether or not to revert mangling from galaxy useful if galaxy produces strange command parameters. Not necessary, but produces more explicit outputs.
args	• The result of commandArgs function, or from the optparse_parameters function. Can be NULL. In this case, uses commandArgs to get the args.
logger	• You can provide a logger to use. If not provided, a logger will be created with the tool's name.
source_files	• You may provide some paths to source before executing the provided code.
env	• You may provide a environment object to execute the code within.
do_traceback	• logical - tells whether to produce a traceback in case of error.

Author(s)

L.Pavot

Examples

```
write_r_file_with_content <- function(content) {
  "
  This function creates a temp R file. It writes the provided
  content in the R file. Then it returns the path of the script.
  "
  path <- tempfile(fileext = ".R")
  file.create(path)
  writeLines(content, con = path)
  return(path)
}
## let's fake a galaxy env
Sys.setenv(GALAXY_SLOTS = 1)
## let's says the tool has been launched with this command line
log_file <- tempfile()
file.create(log_file)
raw_args <- list(
  "--args",
  "--input", "in.csv",
  "--output", "out.csv",
  "--logs", log_file,
  "--one-float", "3.14",
  "--one-integer", "456",
  "--one-logical", "FALSE",
  "--some-floats", "1.5,2.4,3.3",
  "--some-characters", "test,truc,bidule",
```

```
  "--debug", "TRUE",
  "--verbose", "FALSE"
)

##
# example 1
##

my_r_script <- write_r_file_with_content('
  my_processing <- function(args, logger) {
    logger$info("The tool is running")
    logger$infof("Input file: %s.", args$input)
    logger$info("The tool ended.")
  }
')

W4MRUtils::run_galaxy_processing(
  "Test tool 1",
  my_processing(args, logger),
  source_file = my_r_script,
  args = W4MRUtils::parse_args(args = raw_args)
)


##
# example 2

## let's say we have a R script with this content:
path <- write_r_file_with_content('
  setup_logger <- function(args, logger) {
    if (!is.null(args$verbose)) {
      logger$set_verbose(args$verbose)
    }
    if (!is.null(args$debug)) {
      logger$set_debug(args$debug)
    }
    if (!is.null(args$logs)) {
      logger$add_out_paths(args$logs)
    }
  }
  stop_logger <- function(logger) {
    logger$close_files()
  }
  processing <- function(args, logger) {
    setup_logger(args, logger)
    logger$info("The tool is working...")
    logger$infof("Input: %s.", args$input)
    logger$info("The tool stoping.")
    stop_logger(logger)
    return(NULL)
  }')
```

```
## wrapper script:

args <- W4MRUtils::optparse_parameters(
  input = W4MRUtils::optparse_character(),
  output = W4MRUtils::optparse_character(),
  logs = W4MRUtils::optparse_character(),
  one_float = W4MRUtils::optparse_numeric(),
  one_integer = W4MRUtils::optparse_integer(),
  one_logical = W4MRUtils::optparse_flag(),
  some_floats = W4MRUtils::optparse_list(of = "numeric"),
  some_characters = W4MRUtils::optparse_list(of = "character"),
  debug = W4MRUtils::optparse_flag(),
  verbose = W4MRUtils::optparse_flag(),
  args = raw_args[raw_args != "--args"]
)

W4MRUtils::run_galaxy_processing("A Test tool", args = args, {
  ## processing is from the other R script
  processing(args, logger)
}, source_files = path)
```

`show_galaxy_footer` *show_galaxy_footer - shows the footer for glaxy tools*

Description

This function prints the footer to display in galaxy's tools logs

Usage

```
show_galaxy_footer(
  tool_name,
  tool_version,
  logger = NULL,
  show_packages = TRUE,
  ellapsed = NULL
)
```

Arguments

`tool_name`	a `character(1)` holding the running tool's name.
`tool_version`	a `character(1)` holding the running tool's version.
`logger`	a `get_logger("name")` instance - if provided, the galaxy footer if output from the logger.
`show_packages`	logical - Tells whether to display loaded packages and attached packages.
`ellapsed`	NULL or a `character(1)` with execution duration.

Author(s)

L.Pavot

See Also

run_galaxy_processing

Examples

```
show_galaxy_footer("Tool Name", "1.2.0")

show_galaxy_footer(
  tool_name = "Tool Name",
  tool_version = "1.2.0",
  logger = get_logger("Some Tool"),
  show_packages = FALSE,
  ellapsed = "14.5 seconds"
)
```

---

show_galaxy_header *show_galaxy_header - shows the header for glaxy tools*

Description

This function prints the header to display in galaxy's tools logs

Usage

```
show_galaxy_header(
  tool_name,
  tool_version,
  args = NULL,
  logger = NULL,
  show_start_time = TRUE,
  show_sys = TRUE,
  show_parameters = TRUE
)
```

Arguments

tool_name	a character(1) holding the running tool's name.
tool_version	a character(1) holding the running tool's version.
args	a list(param="value") - if provided, their are the parameters shown in galaxy header and/or footer.

`logger` a `get_logger("name")` instance - if provided, the galaxy footer if output from the logger.

`show_start_time`
- a logical telling whether to display the start time or not.

`show_sys`
- a logical telling whether to display the system variables or not.

`show_parameters`
- a logical telling whether to display the parameters or not.

Author(s)

L.Pavot

See Also

run_galaxy_processing

Examples

```
show_galaxy_header("Tool Name", "1.2.0")
show_galaxy_header(
  tool_name = "Tool Name",
  tool_version = "1.2.0",
  logger = get_logger("Some Tool"),
  show_start_time = FALSE,
  show_sys = FALSE,
  show_parameters = FALSE
)
```

---

show_sys *show_sys - prints env variables related to R*

Description

prints env variables related to R

Usage

```
show_sys()
```

Author(s)

L.Pavot

Examples

```
show_sys()
```

shy_lib *Shy Lib*

Description

shy_lib Function to call packages without printing all the verbose (only getting the essentials, like warning messages for example)

Usage

```
shy_lib(...)
```

Arguments

...	Name of libraries to load

Value

a `list` of attached packages

Author(s)

M.Petera

Examples

```
W4MRUtils::shy_lib("base", "utils")
```

source_local *source_local - source file, from absolute or relative path*

Description

source_local Transforms a relative path to an absolute one, and sources the path. This helps source files located relatively to the main script without the need to know from where it was run.

Usage

```
source_local(..., env = FALSE, do_print = FALSE, keep_source = TRUE)
```

Arguments

...	paths, character vector of file paths to source
env	an environement in which to source the paths
do_print	a logical, telling whether to print sourced paths or not
keep_source	See the parameter keep.source from source

Value

a vector resulting from the sourcing of the files provided.

See Also

source()

Examples

```
## let's say we have some R file with the following content:
file_1_content <- "
  setup_logger <- function(args, logger) {
    if (!is.null(args$verbose) && args$verbose) {
      logger$set_verbose(TRUE)
    }
    if (!is.null(args$debug) && args$debug) {
      logger$set_debug(TRUE)
    }
    if (!is.null(args$logs)) {
      logger$add_out_paths(args$logs)
    }
  }"
file_2_content <- "
  processing <- function(args, logger) {
    logger$info(\"The tool is working...\")
    logger$infof(
      \"Parameters: %s\",
      paste(capture.output(str(args)), collapse = \"\n\")
    )
    logger$info(\"The tool ended fine.\")
    return(invisible(NULL))
  }"

if(!file.create(temp_path <- tempfile(fileext = ".R"))) {
  stop("This documentation is not terminated doe to unknown error")
}
writeLines(file_1_content, con = temp_path)

local_path = "test-local-path.R"
local_full_path = file.path(get_base_dir(), local_path)
if(!file.create(local_full_path)) {
  stop("This documentation is not terminated doe to unknown error")
}
writeLines(file_2_content, con = local_full_path)
```

```
## now when we source them, the absolute path is sourced, and the
## relative file path is sourced too.
W4MRUtils::source_local(c(temp_path, local_path), do_print = TRUE)
file.remove(local_full_path)

## the function is accessible here
processing(list(), get_logger("Tool Name"))
```

`stock_id` *Stock ID*

Description

stock_id Functions to stock identifiers before applying make.names() and to reinject it into final matrices. stock_id stocks original identifiers and original order needs checked data regarding table match.

Usage

```
stock_id(data_matrix, metadata, metadata_type)
```

Arguments

`data_matrix`	a `data.frame` containing the data_matrix
`metadata`	a `data.frame` containing samplemetadata or variablemetadata
`metadata_type`	"sample" or "variable" depending on metadata content

Value

a names `list` with three elements:

- id.match a `data.frame` that contains original order of ids, names ;
- dataMatrix the modified data matrix with names sanitized
- Metadata the modified metadata matrix with names sanitized This object can be used in reproduce_id() to replace sanitized names in data matrix by original ones, in the right order.

Author(s)

M.Petera

Examples

```
myDM <- data.frame(data = 1:6, a = 2:7, b = 3:8, c = 2:7, d = 3:8, e = 2:7)
myvM <- data.frame(variable = 1:6, x = 4:9, y = 2:7, z = 3:8)

A <- W4MRUtils::stock_id(myDM, myvM, "variable")
myDM <- A$dataMatrix
myvM <- A$Metadata
A <- A$id.match

## processing that may change order or requires specific identifiers format
## ...
datamatrix <- as.data.frame(myDM)
sample_metadata <- as.data.frame(myvM)

B <- W4MRUtils::reproduce_id(datamatrix, sample_metadata, "variable", A)
datamatrix <- B$dataMatrix
sample_metadata <- B$Metadata
```

`stopaste` *stopaste - to paste string to a message and stop*

Description

stopaste calls paste of its parameters to build the error message and stops with the given message

Usage

```
stopaste(...)
```

Arguments

`...`	Arguments passed on to `base::paste` `sep` a character string to separate the terms. Not `NA_character_`. `collapse` an optional character string to separate the results. Not `NA_character_`. `recycle0` `logical` indicating if zero-length character arguments should lead to the zero-length `character(0)` after the `sep`-phase (which turns into `""` in the `collapse`-phase, i.e., when `collapse` is not `NULL`).

Author(s)

L.Pavot

Examples

```
tryCatch({
  file <- "/tmp/test"
  stopaste("Error in file: ", file)
}, error = function(error) {
  print(error)
})
```

`stopaste0` *stopaste0 - to paste string to a message and stop*

Description

stopaste calls paste0 of its parameters to build the error message and stops with the given message

Usage

```
stopaste0(...)
```

Arguments

`...`	Arguments passed on to `base::paste0`
`collapse`	an optional character string to separate the results. Not `NA_character_`.
`recycle0`	`logical` indicating if zero-length character arguments should lead to the zero-length `character(0)` after the `sep`-phase (which turns into `""` in the `collapse`-phase, i.e., when `collapse` is not `NULL`).

Author(s)

L.Pavot

Examples

```
tryCatch({
  file <- "/tmp/test"
  stopaste0("Error in file: ", file)
}, error = function(error) {
  print(error)
})
```

stopf *stopf - to stop and format message*

Description

stopf calls sprintf of its parameters to build the error message and stops with the given message

Usage

```
stopf(...)
```

Arguments

`...`	Arguments passed on to `base::sprintf` `fmt` a character vector of format strings, each of up to 8192 bytes.

Author(s)

L.Pavot

Examples

```
tryCatch({
  file <- "/tmp/test"
  stopf("Error in %s file.", file)
}, error = function(error) {
  print(error)
})
```

unmangle_galaxy_param *unmangle_galaxy_param - revert effects of galaxy manglings.*

Description

When running a tool from galaxy, the command line may be altered because some forbidden chars have been translated by galaxy.

This function takes `args` are invert the galaxy's mangling process.

Usage

```
unmangle_galaxy_param(args)
```

Arguments

`args`	named `list` - contains params_name=value.

Value

a named `list` - with unmangled parameter name and values.

Author(s)

L.Pavot

See Also

run_galaxy_processing, unmangle_galaxy_string

`unmangle_galaxy_string` *unmangle_galaxy_string - revert effects of galaxy mangling*

Description

Revert effect of string mangling from galaxy on the given string.

Usage

```
unmangle_galaxy_string(string)
```

Arguments

`string`	• the character vector to fix mangling.

Value

string - the character vector, fixed.

Author(s)

L.Pavot

See Also

run_galaxy_processing, unmangle_galaxy_param

`W4MLogger` *The W4MLogger Class*

Description

This is a simple logger used to make uniform outputs across W4M tools.

See get_logger for example usages.

Arguments

`name`	character vector of length 1 - The name of the logger. Use different loggers with a name specific to each part of you program. The name will appear in the log prefix and helps to determine which part of the program did what
`format`	character vector of length 1 - The format string for each log line. The default is : `"[{{ level }}-{{ name }}-{{ time }}] - {{ message }}"`
`do_coloring`	logical vector of length 1 - By default, the logger uses special control character to give some coloring to the text, depending on the log level (info, warning, error, debug or verbose). This coloring is deactivated in files and if `W4MRUtils::in_galaxy_env()` returns TRUE. You can force or deactivate the coloring with this parameter.
`show_debug`	logical vector of length 1 - Tells whether the debug logs must be displayed/written or not. Default is FALSE
`show_verbose`	logical vector of length 1 - Tells whether the verbose logs must be displayed/written or not. Default is FALSE
`show_info`	logical vector of length 1 - Tells whether the info logs must be displayed/written or not Default is TRUE.
`show_warning`	logical vector of length 1 - Tells whether the warning logs must be displayed/written or not Default is TRUE.
`show_error`	logical vector of length 1 - Tells whether the error logs must be displayed/written or not Default is TRUE.
`coloring`	named list - This lists maps a logging level to its coloring. Like this: `list(debug = "purple", info = "green")` Available colors can be found in `W4MRUtils::w4m_colors__`.
`out_func`	function - the default function to print messages in the terminal. The default is `base::message` .
`out_path`	list of file paths - Provide a list of file path where the logs will be written. It is not possible to separate different levels of logs in different log files for the moment.

Value

A W4MLogger instance

Author(s)

L.Pavot

See Also

W4MLogger$info, W4MLogger$warning, W4MLogger$error, W4MLogger$debug, W4MLogger$verbose

---

`W4MLogger_.message__` *W4MLogger_.message__*

Description

The function W4MLogger$.message__ is the function that gets automatically called when W4MLogger$info, W4MLogger$debug, W4MLogger$warning, W4MLogger$error or W4MLogger$verbose are invoked. This function is not truly internal, so it has to be considered as external, but should not be exported:

This means its has to do type checking of its inputs, and consider parameters as unsafe.

See get_logger for example usages.

Arguments

`level`	is a string. By default its value should be either "info", "debug", "warning", "debug", "verbose" or "INTERNAL". But, if the logger was build with a different color naming, one of the names provided in the "coloring" `named list` parameter must be used, as it determines the color to use.
`...`	anything, of any length. If this is not a character vector, then, its displayable value will be obtained with `capture.output(str(...))` If the resulting character vector's length is greater than one, then multiple messages will be printed.

Value

this logger's instance (`.self`)

`W4MLogger_add_out_paths`

Adds a file where logs are duplicated

Description

W4MLogger can output logs in file. This function adds a file in which to put logs.

W4MLogger_finalize *W4MLogger_finalize*

Description

The function W4MLogger$finalize is the destructor function of this class. It closes every files that was opened by the logger, or that was provided during execution. It has to be considered internal.

The function W4MLogger$finalize is the destructor function of this class. It closes every files that was opened by the logger, or that was provided during execution. It has to be considered internal.

The function W4MLogger$finalize is the destructor function of this class. It closes every files that was opened by the logger, or that was provided during execution. It has to be considered internal.

The function W4MLogger$finalize is the destructor function of this class. It closes every files that was opened by the logger, or that was provided during execution. It has to be considered internal.

The function W4MLogger$finalize is the destructor function of this class. It closes every files that was opened by the logger, or that was provided during execution. It has to be considered internal.

W4MLogger_set_debug *W4MLogger$set_debug*

Description

This method activate or deactivate the logging of debugs messages

Arguments

value	logical TRUE/FALSE to activate/deactivate debug logging
default	logical set to TRUE to use debug by default

Value

.self the current W4MLogger instance

`W4MLogger_set_error` *W4MLogger$set_error*

Description

This method activate or deactivate the logging of errors messages

Arguments

`value`	logical TRUE/FALSE to activate/deactivate error logging
`default`	logical set to TRUE to use error by default

Value

.self the current W4MLogger instance

`W4MLogger_set_info` *W4MLogger$set_info*

Description

This method activate or deactivate the logging of info messages

Arguments

`value`	logical TRUE/FALSE to activate/deactivate info logging
`default`	logical set to TRUE to use info by default

Value

.self the current W4MLogger instance

`W4MLogger_set_out_paths`
Defines in which file logs are duplicated

Description

W4MLogger can output logs in file. This function tells in which file to put logs.

W4MLogger_set_verbose *W4MLogger$set_verbose*

Description

This method activate or deactivate the logging of verbose messages

Arguments

value	logical TRUE/FALSE to activate/deactivate verbose logging
default	logical set to TRUE to use verbose by default

Value

.self the current W4MLogger instance

W4MLogger_set_warning *W4MLogger$set_warning*

Description

This method activate or deactivate the logging of warnings messages

Arguments

value	logical TRUE/FALSE to activate/deactivate warning logging
default	logical set to TRUE to use warning by default

Value

.self the current W4MLogger instance

`W4MLogger_[info,warning,error,debug,verbose]` *Log info/warning/error/debug/verbose messages*

Description

Call one of the following function when you want a message to be printed or written in a log file:

- `your_logger$info("A info message");`
- `your_logger$warning("A warning message");`
- `your_logger$error("A error message");`
- `your_logger$debug("A debug message");`
- `your_logger$verbose.("A verbose. message")`

If the corresponding level is activated (with your_logger$set_info(TRUE), your_logger$set_debug(TRUE), etc...), these functions will print the message provided in the terminal and in logs files, if there were some provided at the creation of the logger.

If the corresponding log level is deactivated, these function will not do anything. So, do not hesitate to use them a lot, and activate them when needed.

See get_logger for example usages.

Author(s)

L.Pavot

Index