Package 'assertive.types'

December 30, 2016

Type Package

Title Assertions to Check Types of Variables

Version 0.0-3

Date 2016-12-30

Author Richard Cotton [aut, cre]

Maintainer Richard Cotton <richierocks@gmail.com>

Description A set of predicates and assertions for checking the types of variables. This is mainly for use by other package developers who want to include run-time testing features in their own packages. End-users will usually want to use assertive directly.

URL https://bitbucket.org/richierocks/assertive.types

BugReports https://bitbucket.org/richierocks/assertive.types/issues

Depends R (>= 3.0.0)

Imports assertive.base (>= 0.0-7), assertive.properties, codetools, methods, stats

Suggests testthat, data.table, dplyr, xml2

License GPL (>= 3)

LazyLoad yes

LazyData yes

Acknowledgments Development of this package was partially funded by the Proteomics Core at Weill Cornell Medical College in Qatar <http://qatar-weill.cornell.edu>. The Core is supported by 'Biomedical Research Program' funds, a program funded by Qatar Foundation.

Collate 'imports.R' 'assert-is-a-type.R' 'assert-is-condition.R' 'assert-is-date.R' 'assert-is-formula.R' 'assert-is-function.R' 'assert-is-type-base.R' 'assert-is-type-data.table.R' 'assert-is-type-dplyr.R' 'assert-is-type-grDevices.R' 'assert-is-type-methods.R' 'assert-is-type-stats.R' 'assert-is-type-utils.R' 'is-a-type.R' 'is-condition.R'

'is-date.R' 'is-formula.R' 'is-function.R' 'is-type-base.R'
'is-type-data.table.R' 'is-type-dplyr.R' 'is-type-grDevices.R'
'is-type-methods.R' 'is-type-stats.R' 'is-type-utils.R'

RoxygenNote 5.0.1

ByteCompile true

NeedsCompilation no

Repository CRAN

Date/Publication 2016-12-30 19:35:46

R topics documented:

assert_all_are_classes . . . 3
assert_is_all_of . . . 4
assert_is_an_integer . . . 4
assert_is_array . . . 5
assert_is_a_bool . . . 6
assert_is_a_complex . . . 7
assert_is_a_double . . . 8
assert_is_a_raw . . . 10
assert_is_a_string . . . 11
assert_is_call . . . 12
assert_is_closure_function . . . 13
assert_is_data.frame . . . 14
assert_is_data.table . . . 15
assert_is_date . . . 16
assert_is_environment . . . 17
assert_is_externalptr . . . 18
assert_is_factor . . . 18
assert_is_formula . . . 19
assert_is_function . . . 20
assert_is_inherited_from . . . 21
assert_is_internal_function . . . 22
assert_is_leaf . . . 23
assert_is_list . . . 24
assert_is_mts . . . 25
assert_is_qr . . . 26
assert_is_raster . . . 26
assert_is_relistable . . . 27
assert_is_s3_generic . . . 28
assert_is_S4 . . . 30
assert_is_table . . . 31
assert_is_tbl . . . 32
assert_is_try_error . . . 33

Index **35**

assert_all_are_classes *Is the input the name of a (formally defined) class?*

Description

Checks to see if the input is the name of a (formally defined) class.

Usage

```
assert_all_are_classes(x, severity = getOption("assertive.severity", "stop"))

assert_any_are_classes(x, severity = getOption("assertive.severity", "stop"))

is_class(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_class is a vectorised wrapper for isClass. assert_is_class returns nothing but throws an error if is_class returns FALSE.

See Also

isClass.

Examples

```
assert_all_are_classes(c("lm", "numeric"))
```

assert_is_all_of *Does x belong to these classes?*

Description

Checks to see if x belongs to any of the classes in classes.

Usage

```
assert_is_all_of(x, classes, severity = getOption("assertive.severity",
  "stop"))

assert_is_any_of(x, classes, severity = getOption("assertive.severity",
  "stop"))
```

Arguments

x	Input to check.
classes	As for class.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".

Value

The functions return nothing but throw an error if x does not have any/all of the class classes.

See Also

is2

Examples

```
assert_is_all_of(1:10, c("integer", "numeric"))
#These examples should fail.
assertive.base::dont_stop(assert_is_any_of(1:10, c("list", "data.frame")))
```

assert_is_an_integer *Is the input an integer?*

Description

Checks to see if the input is an integer.

Usage

```
assert_is_an_integer(x, severity = getOption("assertive.severity", "stop"))

assert_is_integer(x, severity = getOption("assertive.severity", "stop"))

is_an_integer(x, .xname = get_name_in_parent(x))

is_integer(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_integer wraps is.integer, providing more information on failure. is_an_integer returns TRUE if the input is an integer and scalar. The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

See Also

is.integer and is_scalar.

Examples

```
assert_is_integer(1:10)
assert_is_an_integer(99L)
#These examples should fail.
assertive.base::dont_stop(assert_is_integer(c(1, 2, 3)))
assertive.base::dont_stop(assert_is_an_integer(1:10))
assertive.base::dont_stop(assert_is_an_integer(integer()))
```

---

assert_is_array *Is the input an array or matrix?*

Description

Checks to see if the input is an array or matrix.

Usage

```
assert_is_array(x, severity = getOption("assertive.severity", "stop"))

assert_is_matrix(x, severity = getOption("assertive.severity", "stop"))

is_array(x, .xname = get_name_in_parent(x))

is_matrix(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_array and is_matrix wrap is.array, and is.matrix respectively, providing more information on failure. The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

Examples

```
assert_is_array(array())
assert_is_array(matrix())
assert_is_matrix(matrix())
#These examples should fail.
assertive.base::dont_stop(assert_is_matrix(array()))
```

---

assert_is_a_bool *Is the input logical?*

Description

Checks to see if the input is logical.

Usage

```
assert_is_a_bool(x, severity = getOption("assertive.severity", "stop"))

assert_is_logical(x, severity = getOption("assertive.severity", "stop"))

is_a_bool(x, .xname = get_name_in_parent(x))

is_logical(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_logical wraps is.logical, providing more information on failure. is_a_bool returns TRUE if the input is logical and scalar. The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

See Also

is.logical and is_scalar.

Examples

```
assert_is_logical(runif(10) > 0.5)
assert_is_a_bool(TRUE)
assert_is_a_bool(NA)
#These examples should fail.
assertive.base::dont_stop(assert_is_logical(1))
assertive.base::dont_stop(assert_is_a_bool(c(TRUE, FALSE)))
assertive.base::dont_stop(assert_is_a_bool(logical()))
```

---

assert_is_a_complex *Is the input complex?*

Description

Checks to see if the input is complex.

Usage

```
assert_is_a_complex(x, severity = getOption("assertive.severity", "stop"))

assert_is_complex(x, severity = getOption("assertive.severity", "stop"))

is_a_complex(x, .xname = get_name_in_parent(x))

is_complex(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_complex wraps is.complex, providing more information on failure. is_a_complex returns TRUE if the input is complex and scalar. The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

See Also

is.complex and is_scalar.

Examples

```
assert_is_complex(c(1i, 2i))
assert_is_a_complex(1i)
assert_is_a_complex(1 + 0i)
assert_is_a_complex(NA_complex_)
#These examples should fail.
assertive.base::dont_stop(assert_is_complex(1:10))
assertive.base::dont_stop(assert_is_a_complex(c(1i, 2i)))
assertive.base::dont_stop(assert_is_a_complex(complex()))
```

---

assert_is_a_double *Is the input numeric?*

Description

Checks to see if the input is numeric.

Usage

```
assert_is_a_double(x, severity = getOption("assertive.severity", "stop"))

assert_is_a_number(x, severity = getOption("assertive.severity", "stop"))

assert_is_double(x, severity = getOption("assertive.severity", "stop"))

assert_is_numeric(x, severity = getOption("assertive.severity", "stop"))

is_a_double(x, .xname = get_name_in_parent(x))

is_a_number(x, .xname = get_name_in_parent(x))
```

```
is_double(x, .xname = get_name_in_parent(x))

is_numeric(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_numeric wraps is.numeric, providing more information on failure. is_a_number returns TRUE if the input is numeric and scalar. The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

Note

numeric means either double or integer, inc this case.

See Also

is_integer, is.numeric and is_scalar.

Examples

```
# "numeric" fns work on double or integers;
assert_is_numeric(1:10)

# Here we check for length 1 as well as type
assert_is_a_number(pi)
assert_is_a_number(1L)
assert_is_a_number(NA_real_)

# "double" fns fail for integers.
assert_is_a_double(pi)

#These examples should fail.
assertive.base::dont_stop(assert_is_numeric(c(TRUE, FALSE)))
assertive.base::dont_stop(assert_is_a_number(1:10))
assertive.base::dont_stop(assert_is_a_number(numeric()))
assertive.base::dont_stop(assert_is_double(1:10))
```

assert_is_a_raw *Is the input raw?*

Description

Checks to see if the input is raw.

Usage

```
assert_is_a_raw(x, severity = getOption("assertive.severity", "stop"))

assert_is_raw(x, severity = getOption("assertive.severity", "stop"))

is_a_raw(x, .xname = get_name_in_parent(x))

is_raw(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_raw wraps is.raw, providing more information on failure. is_a_raw returns TRUE if the input is raw and scalar. The assert_* functions return nothing but throws an error if the corresponding is_* function returns FALSE.

See Also

is.raw and is_scalar.

Examples

```
assert_is_raw(as.raw(1:10))
assert_is_a_raw(as.raw(255))
#These examples should fail.
assertive.base::dont_stop(assert_is_raw(c(TRUE, FALSE)))
assertive.base::dont_stop(assert_is_a_raw(as.raw(1:10)))
assertive.base::dont_stop(assert_is_a_raw(raw()))
```

assert_is_a_string *Is the input of type character?*

Description

Checks to see if the input is of type character.

Usage

```
assert_is_a_string(x, severity = getOption("assertive.severity", "stop"))

assert_is_character(x, severity = getOption("assertive.severity", "stop"))

is_a_string(x, .xname = get_name_in_parent(x))

is_character(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_character wraps is.character, providing more information on failure. is_a_string returns TRUE if the input is character and scalar. The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

See Also

is.character and is_scalar.

Examples

```
assert_is_character(letters)
assertive.base::dont_stop(assert_is_character(factor(letters)))
```

assert_is_call *Is the input a language object?*

Description

Checks to see if the input is a language object.

Usage

```
assert_is_call(x, severity = getOption("assertive.severity", "stop"))

assert_is_expression(x, severity = getOption("assertive.severity", "stop"))

assert_is_language(x, severity = getOption("assertive.severity", "stop"))

assert_is_name(x, severity = getOption("assertive.severity", "stop"))

assert_is_symbol(x, severity = getOption("assertive.severity", "stop"))

is_call(x, .xname = get_name_in_parent(x))

is_expression(x, .xname = get_name_in_parent(x))

is_language(x, .xname = get_name_in_parent(x))

is_name(x, .xname = get_name_in_parent(x))

is_symbol(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_call, is_expression, is_language, is_name and is_symbol wrap the corresponding is.* functions, providing more information on failure. The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

Note

is_name and is_symbol are different names for the same function.

See Also

is.call, is.expression is.language and is.name.

Examples

```
a_call <- call("sin", "pi")
assert_is_call(a_call)
assert_is_language(a_call)
an_expression <- expression(sin(pi))
assert_is_expression(an_expression)
assert_is_language(an_expression)
a_name <- as.name("foo")
assert_is_name(a_name)
assert_is_language(a_name)
#These examples should fail.
assertive.base::dont_stop(assert_is_language(function(){}))
```

---

assert_is_closure_function

Is the input a closure, builtin or special function?

Description

Checks to see if the input is a closure, builtin or special function.

Usage

```
assert_is_closure_function(x, severity = getOption("assertive.severity",
  "stop"))

assert_is_builtin_function(x, severity = getOption("assertive.severity",
  "stop"))

assert_is_special_function(x, severity = getOption("assertive.severity",
  "stop"))

is_closure_function(x, .xname = get_name_in_parent(x))

is_builtin_function(x, .xname = get_name_in_parent(x))

is_special_function(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

`is_internal_function` returns `TRUE` when the input is a closure function that calls `.Internal`. The `assert_*` function returns nothing but throw an error if the corresponding `is_*` function returns `FALSE`.

References

There is some discussion of closure vs. builtin vs. special functions in the Argument Evaluation section of R-internals. https://cran.r-project.org/doc/manuals/r-devel/R-ints.html#Argument-evaluation

See Also

`is.function` and its assertive wrapper `is_function`. `typeof` is used to distinguish the three types of function.

Examples

```
# most functions are closures
is_closure_function(mean)
is_closure_function(lm)
is_closure_function(summary)

# builtin functions are typically math operators, low level math functions
# and commonly used functions
is_builtin_function(`*`)
is_builtin_function(cumsum)
is_builtin_function(is.numeric)

# special functions are mostly language features
is_special_function(`if`)
is_special_function(`return`)
is_special_function(`~`)

# some failure messages
assertive.base::dont_stop({
assert_is_builtin_function(mean)
assert_is_builtin_function("mean")
})
```

---

assert_is_data.frame *Is the input is a data.frame?*

Description

Is the input is a data.frame?

Usage

```
assert_is_data.frame(x, severity = getOption("assertive.severity", "stop"))

is_data.frame(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_data.frame wraps is.data.frame, providing more information on failure. assert_is_data.frame returns nothing but throws an error if is_data.frame returns FALSE.

See Also

is.data.frame.

Examples

```
assert_is_data.frame(data.frame())
assert_is_data.frame(datasets::CO2)
```

---

assert_is_data.table *Is the input a data.table?*

Description

Checks to see if the input is a data.table.

Usage

```
assert_is_data.table(x, severity = getOption("assertive.severity", "stop"))

is_data.table(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

`is_data.table` wraps `is.data.table`, providing more information on failure. The `assert_*` functions return nothing but throws an error if the corresponding `is_*` function returns `FALSE`.

See Also

`is.data.table`.

Examples

```
if(requireNamespace("data.table"))
{
  assert_is_data.table(data.table::data.table(x = 1:5))
  #These examples should fail.
  assertive.base::dont_stop(assert_is_data.table(list(1,2,3)))
} else
{
  message("This example requires the data.table package to be installed.")
}
```

---

assert_is_date *Is the input a date?*

Description

Checks to see if the input is a Date or POSIXt object.

Usage

```
assert_is_date(x, severity = getOption("assertive.severity", "stop"))

assert_is_posixct(x, severity = getOption("assertive.severity", "stop"))

assert_is_posixlt(x, severity = getOption("assertive.severity", "stop"))

is_date(x, .xname = get_name_in_parent(x))

is_posixct(x, .xname = get_name_in_parent(x))

is_posixlt(x, .xname = get_name_in_parent(x))
```

Arguments

`x`	Input to check.
`severity`	How severe should the consequences of the assertion be? Either `"stop"`, `"warning"`, `"message"`, or `"none"`.
`.xname`	Not intended to be used directly.

Value

The `is_*` functions return `TRUE` or `FALSE` depending upon whether or not the input is a datetime object.

The `assert_*` functions return nothing but throw an error if the corresponding `is_*` function returns `FALSE`.

Examples

```
is_date(Sys.Date())
is_posixct(Sys.time())

# These examples should fail.
assertive.base::dont_stop(assert_is_date(Sys.time()))
```

---

assert_is_environment *Is the input an environment?*

Description

Checks to see if the input is an environment.

Usage

```
assert_is_environment(x, severity = getOption("assertive.severity", "stop"))

is_environment(x, .xname = get_name_in_parent(x))
```

Arguments

`x`	Input to check.
`severity`	How severe should the consequences of the assertion be? Either `"stop"`, `"warning"`, `"message"`, or `"none"`.
`.xname`	Not intended to be used directly.

Value

`is_environment` wraps `is.environment`, providing more information on failure. `assert_is_environment` returns nothing but throws an error if `is_environment` returns `FALSE`.

See Also

`is.environment`.

Examples

```
assert_is_environment(new.env())
assert_is_environment(globalenv())
assert_is_environment(baseenv())
```

assert_is_externalptr *Is the input is an external pointer?*

Description

Check whether the input is an external pointer. that is, an object of class ("externalptr").

Usage

```
assert_is_externalptr(x, severity = getOption("assertive.severity", "stop"))

is_externalptr(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_externalptr wraps is.data.frame, providing more information on failure. assert_is_externalptr returns nothing but throws an error if is_externalptr returns FALSE.

Examples

```
# The xml2 pkg makes heavy use of external pointers
xptr <- xml2::read_xml("<foo><bar /></foo>")$node
assert_is_externalptr(xptr)

# This should fail
assertive.base::dont_stop({
assert_is_externalptr(NULL)
})
```

assert_is_factor *Is the input a factor?*

Description

Checks to see if the input is an factor.

Usage

```
assert_is_factor(x, severity = getOption("assertive.severity", "stop"))

assert_is_ordered(x, severity = getOption("assertive.severity", "stop"))

is_factor(x, .xname = get_name_in_parent(x))

is_ordered(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_factor wraps is.factor, providing more information on failure. assert_is_factor returns nothing but throws an error if is_factor returns FALSE.

See Also

is.factor.

Examples

```
assert_is_factor(factor(sample(letters, 10)))
```

---

assert_is_formula *Is the input a formula?*

Description

Checks to see if the input is a formula.

Usage

```
assert_is_formula(x, severity = getOption("assertive.severity", "stop"))

assert_is_one_sided_formula(x, severity = getOption("assertive.severity",
  "stop"))

assert_is_two_sided_formula(x, severity = getOption("assertive.severity",
  "stop"))

is_formula(x, .xname = get_name_in_parent(x))
```

```
is_one_sided_formula(x, .xname = get_name_in_parent(x))

is_two_sided_formula(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

The is_* functions return TRUE when the input is a formula. The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

See Also

is_environment and is_language

Examples

```
is_one_sided_formula(~ x)
is_two_sided_formula(y ~ x)
```

---

assert_is_function *Is the input a function?*

Description

Checks to see if the input is a function.

Usage

```
assert_is_function(x, severity = getOption("assertive.severity", "stop"))

assert_is_primitive(x, severity = getOption("assertive.severity", "stop"))

assert_is_stepfun(x, severity = getOption("assertive.severity", "stop"))

is_function(x, .xname = get_name_in_parent(x))

is_primitive(x, .xname = get_name_in_parent(x))

is_stepfun(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_function, is_primitive and is_stepfun wrap is.function, is.primitive and is.stepfun repsectively, providing more information on failure. The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

See Also

is.function.

Examples

```
assert_is_function(sqrt)
assert_is_function(function(){})
```

---

assert_is_inherited_from

Does the object inherit from some class?

Description

Checks to see if an object is inherited from any of the specified classes.

Usage

```
assert_is_inherited_from(x, classes,
  severity = getOption("assertive.severity", "stop"))

is_inherited_from(x, classes, .xname = get_name_in_parent(x))
```

Arguments

x	Any R variable.
classes	A character vector of classes.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

TRUE if x inherits from at least one of the classes, as determined by inherits.

See Also

inherits, is, is2

Examples

```
x <- structure(1:5, class = c("foo", "bar"))
assert_is_inherited_from(x, c("foo", "baz"))
assertive.base::dont_stop(assert_is_inherited_from(x, c("Foo", "baz")))
```

---

assert_is_internal_function

Is the input an internal function?

Description

Checks to see if the input is an internal function. That is, it is a non-primitive function that calls C-code via .Internal.

Usage

```
assert_is_internal_function(x, severity = getOption("assertive.severity",
  "stop"))

is_internal_function(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_internal_function returns TRUE when the input is a closure function that calls .Internal. The assert_* function returns nothing but throw an error if the corresponding is_* function returns FALSE.

References

This function is modeled upon is_internal, internal to the pryr package. The differences between the .Internal and .Primitive interfaces to C code are discussed in R-Internals, in the chapter Internal vs. Primitive. https://cran.r-project.org/doc/manuals/r-devel/R-ints.html#g_t_002eInternal-vs-_002ePrimitive

See Also

is.function and its assertive wrapper is_function.

Examples

```
# Some common fns calling .Internal
is_internal_function(unlist)
is_internal_function(cbind)

# Some failures
assertive.base::dont_stop({
assert_is_internal_function("unlist")
assert_is_internal_function(sqrt)
assert_is_internal_function(function(){})
})
```

---

assert_is_leaf *Is the input a (dendrogram) leaf?*

Description

Checks to see if the input is a (dendrogram) leaf.

Usage

```
assert_is_leaf(x, severity = getOption("assertive.severity", "stop"))

is_leaf(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_leaf reimplements is.leaf, providing more information on failure.

See Also

dendrogram.

`assert_is_list` *Is the input a list?*

Description

Checks to see if the input is a list.

Usage

```
assert_is_list(x, severity = getOption("assertive.severity", "stop"))

assert_is_pairlist(x, severity = getOption("assertive.severity", "stop"))

is_list(x, .xname = get_name_in_parent(x))

is_pairlist(x, .xname = get_name_in_parent(x))
```

Arguments

`x`	Input to check.
`severity`	How severe should the consequences of the assertion be? Either `"stop"`, `"warning"`, `"message"`, or `"none"`.
`.xname`	Not intended to be used directly.

Value

`is_list` wraps `is.list`, providing more information on failure.

See Also

`is.list`.

Examples

```
assert_is_list(list(1,2,3))
assert_is_pairlist(.Options)
#These examples should fail.
assertive.base::dont_stop({
  assert_is_list(1:10)
  assert_is_pairlist(options())
})
```

assert_is_mts *Is the input a time series?*

Description

Checks to see if the input is a time series.

Usage

```
assert_is_mts(x, severity = getOption("assertive.severity", "stop"))

assert_is_ts(x, severity = getOption("assertive.severity", "stop"))

assert_is_tskernel(x, severity = getOption("assertive.severity", "stop"))

is_mts(x, .xname = get_name_in_parent(x))

is_ts(x, .xname = get_name_in_parent(x))

is_tskernel(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_ts wraps is.ts, providing more information on failure. assert_is_ts returns nothing but throws an error if is_ts returns FALSE.

See Also

is.ts.

Examples

```
assert_is_ts(ts(1:10))
```

assert_is_qr *Is the input a QR decomposition of a matrix?*

Description

Checks to see if the input is a QR decomposition of a matrix.

Usage

```
assert_is_qr(x, severity = getOption("assertive.severity", "stop"))

is_qr(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_qr wraps is.qr, providing more information on failure. assert_is_qr returns nothing but throws an error if is_qr returns FALSE.

See Also

is.qr.

Examples

```
assert_is_qr(qr(matrix(rnorm(25), nrow = 5)))
```

assert_is_raster *Is the input a raster?*

Description

Checks to see if the input is a raster.

Usage

```
assert_is_raster(x, severity = getOption("assertive.severity", "stop"))

is_raster(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_raster wraps is.raster, providing more information on failure. is_a_raster returns TRUE if the input is raster and scalar. The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

See Also

is.raster.

Examples

```
m <- matrix(hcl(0, 80, seq(50, 80, 10)), nrow=4, ncol=5)
assert_is_raster(as.raster(m))
## Not run:
#These examples should fail.
assert_is_raster(m)

## End(Not run)
```

---

assert_is_relistable *Is the input relistable?*

Description

Checks to see if the input is relistable.

Usage

```
assert_is_relistable(x, severity = getOption("assertive.severity", "stop"))

is_relistable(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_relistable wraps is.relistable, providing more information on failure. The assert_* functions return nothing but throws an error if the corresponding is_* function returns FALSE.

See Also

is.relistable and is_scalar.

Examples

```
assert_is_relistable(as.relistable(list(1,2,3)))
#These examples should fail.
assertive.base::dont_stop(assert_is_relistable(list(1,2,3)))
```

---

assert_is_s3_generic *Is the input an S3 generic or method?*

Description

Checks whether the input is an S3 generic or method.

Usage

```
assert_is_s3_generic(x, severity = getOption("assertive.severity", "stop"))

assert_is_s3_method(x, severity = getOption("assertive.severity", "stop"))

assert_is_s3_primitive_generic(x, severity = getOption("assertive.severity",
  "stop"))

assert_is_s3_group_generic(x, severity = getOption("assertive.severity",
  "stop"))

assert_is_s4_group_generic(x, severity = getOption("assertive.severity",
  "stop"))

assert_is_s3_internal_generic(x, severity = getOption("assertive.severity",
  "stop"))

is_s3_generic(x, .xname = get_name_in_parent(x))

is_s3_method(x, .xname = get_name_in_parent(x))

is_s3_primitive_generic(x, .xname = get_name_in_parent(x))

is_s3_group_generic(x, .xname = get_name_in_parent(x))
```

```
is_s4_group_generic(x, groups = c("Arith", "Compare", "Ops", "Logic", "Math",
  "Math2", "Summary", "Complex"), .xname = get_name_in_parent(x))

is_s3_internal_generic(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.
groups	A character vector of S4 group generic groups.

Value

is_internal_function returns TRUE when the input is a closure function that calls .Internal. The assert_* function returns nothing but throw an error if the corresponding is_* function returns FALSE.

References

is_s3_generic is based upon is_s3_generic. Similarly, is_s3_method is based upon find_generic, internal to pryr, with some ideas from isS3method. is_primitive_generic checks for the functions listed by .S3PrimitiveGenerics. is_s3_group_generic checks for the functions listed by .get_internal_S3_generics, internal to the tools package. is_s4_group_generic checks for the functions listed by getGroupMembers. S4 group generics are mostly the same as S3 group generics, except that the not operator, !, is S3 group generic but not S4, and log2 and log10 are S4 group generic but not S3. is_s3_internal_generic checks for the functions listed by .get_internal_S3_generics, internal to the tools package. internal_generics, internal to pryr works similarly, though checks for S4 group generics rather than S3 group generics. There is some discussion of group generics scatterd throughout R-internals. In particular, see the section on the Mechanics of S4 Dispatch. https://cran.r-project.org/doc/manuals/r-devel/R-ints.html#Mechanics-of-S4-dispatch

See Also

is.function and its assertive wrapper is_function. is_closure_function to check for closures/builtin and special functions. is_internal_function to check for functions that use the .Internal interface to C code.

Examples

```
# General check for S3 generics and methods
is_s3_generic(is.na)
is_s3_method(is.na.data.frame)

# More specific types of S3 generic
is_s3_primitive_generic(c)
is_s3_group_generic(abs)
```

```
is_s3_internal_generic(unlist)

# S4 group generics are mostly the same as S3 group generics
is_s4_group_generic(cosh)

# Renaming functions is fine
not <- `!`
is_s3_group_generic(not)

# Some failures
assertive.base::dont_stop({
assert_is_s3_primitive_generic(exp)
assert_is_s4_group_generic(`!`)
})
```

`assert_is_S4` *Is the input an S4 object?*

Description

Checks to see if the input is an S4 object.

Usage

```
assert_is_S4(x, severity = getOption("assertive.severity", "stop"))

assert_is_s4(x, severity = getOption("assertive.severity", "stop"))

assert_is_ref_class_generator(x, severity = getOption("assertive.severity",
  "stop"))

assert_is_ref_class_object(x, severity = getOption("assertive.severity",
  "stop"))

is_s4(x, .xname = get_name_in_parent(x))

is_S4(x, .xname = get_name_in_parent(x))

is_ref_class_generator(x, .xname = get_name_in_parent(x))

is_ref_class_object(x, .xname = get_name_in_parent(x))
```

Arguments

`x`	Input to check.
`severity`	How severe should the consequences of the assertion be? Either `"stop"`, `"warning"`, `"message"`, or `"none"`.
`.xname`	Not intended to be used directly.

Value

is_S4 wraps isS4, providing more information on failure. assert_is_S4 returns nothing but throws an error if is_S4 returns FALSE.

See Also

isS4.

Examples

```
assert_is_s4(getClass("MethodDefinition"))
# These examples should fail.
assertive.base::dont_stop(assert_is_s4(1:10))
```

---

assert_is_table *Is the input a table?*

Description

Checks to see if the input is a table.

Usage

```
assert_is_table(x, severity = getOption("assertive.severity", "stop"))

is_table(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_table wraps is.table, providing more information on failure. assert_is_table returns nothing but throws an error if is_table returns FALSE.

See Also

is.table.

Examples

```
assert_is_table(table(sample(letters, 100, replace = TRUE)))
```

assert_is_tbl *Is the input a tbl?*

Description

Checks to see if the input is a tbl.

Usage

```
assert_is_tbl(x, severity = getOption("assertive.severity", "stop"))

assert_is_tbl_cube(x, severity = getOption("assertive.severity", "stop"))

assert_is_tbl_df(x, severity = getOption("assertive.severity", "stop"))

assert_is_tbl_dt(x, severity = getOption("assertive.severity", "stop"))

is_tbl(x, .xname = get_name_in_parent(x))

is_tbl_cube(x, .xname = get_name_in_parent(x))

is_tbl_df(x, .xname = get_name_in_parent(x))

is_tbl_dt(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

is_data.table wraps is.data.table, providing more information on failure. The assert_* functions return nothing but throws an error if the corresponding is_* function returns FALSE.

See Also

is.data.table.

Examples

```
if(requireNamespace("dplyr"))
{
  assert_is_tbl_df(dplyr::tbl_df(data.frame(x = 1:5)))
  #These examples should fail.
```

```
  assertive.base::dont_stop(assert_is_tbl(data.frame(x = 1:5)))
} else
{
  message("This example requires the data.table package to be installed.")
}
```

---

assert_is_try_error *Is the input a condition?*

Description

Checks to see if the input is a message, warning or error.

Usage

```
assert_is_try_error(x, severity = getOption("assertive.severity", "stop"))

assert_is_simple_error(x, severity = getOption("assertive.severity", "stop"))

assert_is_error(x, severity = getOption("assertive.severity", "stop"))

assert_is_simple_warning(x, severity = getOption("assertive.severity",
  "stop"))

assert_is_warning(x, severity = getOption("assertive.severity", "stop"))

assert_is_simple_message(x, severity = getOption("assertive.severity",
  "stop"))

assert_is_message(x, severity = getOption("assertive.severity", "stop"))

assert_is_condition(x, severity = getOption("assertive.severity", "stop"))

is_try_error(x, .xname = get_name_in_parent(x))

is_simple_error(x, .xname = get_name_in_parent(x))

is_error(x, .xname = get_name_in_parent(x))

is_simple_warning(x, .xname = get_name_in_parent(x))

is_warning(x, .xname = get_name_in_parent(x))

is_simple_message(x, .xname = get_name_in_parent(x))

is_message(x, .xname = get_name_in_parent(x))

is_condition(x, .xname = get_name_in_parent(x))
```

Arguments

x	Input to check.
severity	How severe should the consequences of the assertion be? Either "stop", "warning", "message", or "none".
.xname	Not intended to be used directly.

Value

The is_* functions return TRUE or FALSE depending upon whether or not the input is a datetime object.

The assert_* functions return nothing but throw an error if the corresponding is_* function returns FALSE.

Examples

```
# stop returns simpleErrors, unless wrapped in a call to try
simple_err <- tryCatch(stop("!!!"), error = function(e) e)
is_simple_error(simple_err)
try_err <- try(stop("!!!"))
is_try_error(try_err)

# is_error checks for both error types
is_error(try_err)
is_error(simple_err)

# warning returns simpleWarnings
simple_warn <- tryCatch(warning("!!!"), warning = function(w) w)
is_simple_warning(simple_warn)
is_warning(simple_warn)

# message returns simpleMessages
simple_msg <- tryCatch(message("!!!"), message = function(m) m)
is_simple_message(simple_msg)
is_message(simple_msg)

# These examples should fail.
assertive.base::dont_stop(assert_is_simple_error(try_err))
assertive.base::dont_stop(assert_is_try_error(simple_err))
```

Index

.Internal, *14*, *22*, *29*
.S3PrimitiveGenerics, *29*

assert_all_are_classes, 3
assert_any_are_classes
(assert_all_are_classes), 3
assert_is_a_bool, 6
assert_is_a_complex, 7
assert_is_a_double, 8
assert_is_a_number
(assert_is_a_double), 8
assert_is_a_raw, 10
assert_is_a_string, 11
assert_is_all_of, 4
assert_is_an_integer, 4
assert_is_any_of (assert_is_all_of), 4
assert_is_array, 5
assert_is_builtin_function
(assert_is_closure_function), 13
assert_is_call, 12
assert_is_character
(assert_is_a_string), 11
assert_is_closure_function, 13
assert_is_complex
(assert_is_a_complex), 7
assert_is_condition
(assert_is_try_error), 33
assert_is_data.frame, 14
assert_is_data.table, 15
assert_is_date, 16
assert_is_double (assert_is_a_double), 8
assert_is_environment, 17
assert_is_error (assert_is_try_error), 33
assert_is_expression (assert_is_call), 12
assert_is_externalptr, 18
assert_is_factor, 18
assert_is_formula, 19
assert_is_function, 20
assert_is_inherited_from, 21
assert_is_integer
(assert_is_an_integer), 4
assert_is_internal_function, 22
assert_is_language (assert_is_call), 12
assert_is_leaf, 23
assert_is_list, 24
assert_is_logical (assert_is_a_bool), 6
assert_is_matrix (assert_is_array), 5
assert_is_message
(assert_is_try_error), 33
assert_is_mts, 25
assert_is_name (assert_is_call), 12
assert_is_numeric (assert_is_a_double), 8
assert_is_one_sided_formula
(assert_is_formula), 19
assert_is_ordered (assert_is_factor), 18
assert_is_pairlist (assert_is_list), 24
assert_is_posixct (assert_is_date), 16
assert_is_posixlt (assert_is_date), 16
assert_is_primitive
(assert_is_function), 20
assert_is_qr, 26
assert_is_raster, 26
assert_is_raw (assert_is_a_raw), 10
assert_is_ref_class_generator
(assert_is_S4), 30
assert_is_ref_class_object
(assert_is_S4), 30
assert_is_relistable, 27
assert_is_s3_generic, 28
assert_is_s3_group_generic
(assert_is_s3_generic), 28
assert_is_s3_internal_generic
(assert_is_s3_generic), 28
assert_is_s3_method
(assert_is_s3_generic), 28

assert_is_s3_primitive_generic
(assert_is_s3_generic), 28
assert_is_S4, 30
assert_is_s4 (assert_is_S4), 30
assert_is_s4_group_generic
(assert_is_s3_generic), 28
assert_is_simple_error
(assert_is_try_error), 33
assert_is_simple_message
(assert_is_try_error), 33
assert_is_simple_warning
(assert_is_try_error), 33
assert_is_special_function
(assert_is_closure_function),
13
assert_is_stepfun (assert_is_function),
20
assert_is_symbol (assert_is_call), 12
assert_is_table, 31
assert_is_tbl, 32
assert_is_tbl_cube (assert_is_tbl), 32
assert_is_tbl_df (assert_is_tbl), 32
assert_is_tbl_dt (assert_is_tbl), 32
assert_is_try_error, 33
assert_is_ts (assert_is_mts), 25
assert_is_tskernel (assert_is_mts), 25
assert_is_two_sided_formula
(assert_is_formula), 19
assert_is_warning
(assert_is_try_error), 33

dendrogram, *23*

getGroupMembers, *29*

inherits, *21, 22*
is, *22*
is.call, *13*
is.character, *11*
is.complex, *8*
is.data.frame, *15*
is.data.table, *16*, *32*
is.environment, *17*
is.expression, *13*
is.factor, *19*
is.function, *14*, *21, 22*, *29*
is.integer, *5*
is.language, *13*
is.list, *24*
is.logical, *7*
is.name, *13*
is.numeric, *9*
is.qr, *26*
is.raster, *27*
is.raw, *10*
is.relistable, *28*
is.table, *31*
is.ts, *25*
is2, *4*, *22*
is_a_bool (assert_is_a_bool), 6
is_a_complex (assert_is_a_complex), 7
is_a_double (assert_is_a_double), 8
is_a_number (assert_is_a_double), 8
is_a_raw (assert_is_a_raw), 10
is_a_string (assert_is_a_string), 11
is_an_integer (assert_is_an_integer), 4
is_array (assert_is_array), 5
is_builtin_function
(assert_is_closure_function),
13
is_call (assert_is_call), 12
is_character (assert_is_a_string), 11
is_class (assert_all_are_classes), 3
is_closure_function, *29*
is_closure_function
(assert_is_closure_function),
13
is_complex (assert_is_a_complex), 7
is_condition (assert_is_try_error), 33
is_data.frame (assert_is_data.frame), 14
is_data.table (assert_is_data.table), 15
is_date (assert_is_date), 16
is_double (assert_is_a_double), 8
is_environment, *20*
is_environment (assert_is_environment),
17
is_error (assert_is_try_error), 33
is_expression (assert_is_call), 12
is_externalptr (assert_is_externalptr),
18
is_factor (assert_is_factor), 18
is_formula (assert_is_formula), 19
is_function, *14*, *22*, *29*
is_function (assert_is_function), 20
is_inherited_from
(assert_is_inherited_from), 21
is_integer, *9*

is_integer *(assert_is_an_integer)*, 4
is_internal_function, *29*
is_internal_function
(assert_is_internal_function), 22
is_language, *20*
is_language *(assert_is_call)*, 12
is_leaf *(assert_is_leaf)*, 23
is_list *(assert_is_list)*, 24
is_logical *(assert_is_a_bool)*, 6
is_matrix *(assert_is_array)*, 5
is_message *(assert_is_try_error)*, 33
is_mts *(assert_is_mts)*, 25
is_name *(assert_is_call)*, 12
is_numeric *(assert_is_a_double)*, 8
is_one_sided_formula
(assert_is_formula), 19
is_ordered *(assert_is_factor)*, 18
is_pairlist *(assert_is_list)*, 24
is_posixct *(assert_is_date)*, 16
is_posixlt *(assert_is_date)*, 16
is_primitive *(assert_is_function)*, 20
is_qr *(assert_is_qr)*, 26
is_raster *(assert_is_raster)*, 26
is_raw *(assert_is_a_raw)*, 10
is_ref_class_generator *(assert_is_S4)*, 30
is_ref_class_object *(assert_is_S4)*, 30
is_relistable *(assert_is_relistable)*, 27
is_s3_generic, *29*
is_s3_generic *(assert_is_s3_generic)*, 28
is_s3_group_generic
(assert_is_s3_generic), 28
is_s3_internal_generic
(assert_is_s3_generic), 28
is_s3_method *(assert_is_s3_generic)*, 28
is_s3_primitive_generic
(assert_is_s3_generic), 28
is_S4 *(assert_is_S4)*, 30
is_s4 *(assert_is_S4)*, 30
is_s4_group_generic
(assert_is_s3_generic), 28
is_scalar, *5*, *7–11*, *28*
is_simple_error *(assert_is_try_error)*, 33
is_simple_message
(assert_is_try_error), 33
is_simple_warning
(assert_is_try_error), 33
is_special_function
(assert_is_closure_function), 13
is_stepfun *(assert_is_function)*, 20
is_symbol *(assert_is_call)*, 12
is_table *(assert_is_table)*, 31
is_tbl *(assert_is_tbl)*, 32
is_tbl_cube *(assert_is_tbl)*, 32
is_tbl_df *(assert_is_tbl)*, 32
is_tbl_dt *(assert_is_tbl)*, 32
is_try_error *(assert_is_try_error)*, 33
is_ts *(assert_is_mts)*, 25
is_tskernel *(assert_is_mts)*, 25
is_two_sided_formula
(assert_is_formula), 19
is_warning *(assert_is_try_error)*, 33
isClass, *3*
isS3method, *29*
isS4, *31*

typeof, *14*