Package 'validatetools'

May 8, 2026

Title Checking and Simplifying Validation Rule Sets

Version 0.6.1

Description Rule sets with validation rules may contain redundancies or contradictions.
Functions for finding redundancies and problematic rules are provided,
given a set a rules formulated with 'validate'.

Depends validate, R (>= 4.1.0)

License MIT + file LICENSE

Encoding UTF-8

URL https://github.com/data-cleaning/validatetools

BugReports https://github.com/data-cleaning/validatetools/issues

Imports methods, stats, utils, lpSolveAPI

Suggests testthat, covr

RoxygenNote 7.3.2

NeedsCompilation no

Author Edwin de Jonge [aut, cre] (ORCID:
<https://orcid.org/0000-0002-6580-4718>),
Mark van der Loo [aut],
Jacco Daalmans [ctb]

Maintainer Edwin de Jonge <edwindjonge@gmail.com>

Repository CRAN

Date/Publication 2025-07-11 12:50:02 UTC

Contents

detect_boundary_cat 2
detect_boundary_num 3
detect_contradicting_if_rules 4
detect_fixed_variables 5
detect_infeasible_rules 6
detect_redundancy 7

is_categorical 8
is_conditional 9
is_contradicted_by 10
is_implied_by 11
is_infeasible 12
is_linear 13
make_feasible 13
remove_redundancy 15
simplify_conditional 16
simplify_fixed_variables 17
simplify_rules 18
substitute_values 19
validatetools 20

Index **22**

`detect_boundary_cat` *Detect domains for categorical variables*

Description

Detect the allowed values for categorical variables: the rule set may constrain the categorical variables to a subset of their values. `detect_boundary_cat()` finds the categories that are allowed by the rule set.

Usage

```
detect_boundary_cat(x, ..., as_df = FALSE)
```

Arguments

`x`	`validate::validator()` object with rules
`...`	not used
`as_df`	return result as data.frame (before 0.4.5)

Value

`data.frame` with columns `$variable`, `$value`, `$min`, `$max`. Each row is a category/value of a categorical variable.

See Also

Other feasibility: `detect_boundary_num()`, `detect_contradicting_if_rules()`, `detect_infeasible_rules()`, `is_contradicted_by()`, `is_infeasible()`, `make_feasible()`

Examples

```
rules <- validator(
  x >= 1,
  x + y <= 10,
  y >= 6
)

detect_boundary_num(rules)

rules <- validator(
  job %in% c("yes", "no"),
  if (job == "no") income == 0,
  income > 0
)

detect_boundary_cat(rules)
```

`detect_boundary_num` *Detect the allowed range for numerical variables*

Description

Detect for each numerical variable in a validation rule set, what its maximum and minimum allowed values are. The rule set may constrain numerical variables to a subset of their values. This allows for manual rule set checking: does rule set `x` overly constrain numerical values?

Usage

```
detect_boundary_num(x, eps = 1e-08, ...)
```

Arguments

`x`	`validate::validator()` object, rule set to be checked
`eps`	detected fixed values will have this precision.
`...`	currently not used

Details

This procedure only finds minimum and maximum values, but misses gaps.

Value

`data.frame()` with columns "variable", "lowerbound", "upperbound".

References

Statistical Data Cleaning with R (2017), Chapter 8, M. van der Loo, E. de Jonge

Simplifying constraints in data editing (2015). Technical Report 2015|18, Statistics Netherlands, J. Daalmans

See Also

detect_fixed_variables()

Other feasibility: detect_boundary_cat(), detect_contradicting_if_rules(), detect_infeasible_rules(), is_contradicted_by(), is_infeasible(), make_feasible()

Examples

```
rules <- validator(
  x >= 1,
  x + y <= 10,
  y >= 6
)

detect_boundary_num(rules)

rules <- validator(
  job %in% c("yes", "no"),
  if (job == "no") income == 0,
  income > 0
)

detect_boundary_cat(rules)
```

---

detect_contradicting_if_rules

Detect contradictory if-rules

Description

Detect whether conditions in conditional if-rules may generate contradictions. Strictly speaking these rules do not make the rule set infeasible but rather make the if-condition unsatisfiable. Semantically speaking these rules are contradicting, because the writer of the rule set did not have the intention to make the condition forbidden.

Usage

```
detect_contradicting_if_rules(x, ..., verbose = interactive())
```

Arguments

x	A validator object.
...	Additional arguments passed to detect_if_clauses.
verbose	Logical. If TRUE, print the results.

Details

In general it detects (variations on) cases where:

- `if (A) B` and `if (A) !B`, which probably is not intended, but logically equals `!A`.
- `if (A) B` and `if (B) !A`, which probably is not intended but logically equals `!A`.

See examples for more details.

Value

A list of contradictions found in the if clauses, or `NULL` if none are found.

See Also

Other feasibility: `detect_boundary_cat()`, `detect_boundary_num()`, `detect_infeasible_rules()`, `is_contradicted_by()`, `is_infeasible()`, `make_feasible()`

Examples

```
rules <- validator(
  if (nace == "a") export == "y",
  if (nace == "a")  export == "n"
)

conflicts <- detect_contradicting_if_rules(rules, verbose=TRUE)

print(conflicts)


# this creates a implicit contradiction when income > 0
rules <- validator(
  rule1 = if (income > 0) job == "yes",
  rule2 = if (job == "yes") income == 0
)

conflicts <- detect_contradicting_if_rules(rules, verbose=TRUE)
```

`detect_fixed_variables` *Detect fixed variables*

Description

Detects variables that are constrained by the rule set to have one fixed value. To simplify a rule set, these variables can be substituted with their value. See examples.

Usage

```
detect_fixed_variables(x, eps = x$options("lin.eq.eps"), ...)
```

Arguments

x	validate::validator() object with the validation rules.
eps	detected fixed values will have this precission.
...	not used.

See Also

simplify_fixed_variables()

Other redundancy: detect_redundancy(), is_implied_by(), remove_redundancy(), simplify_fixed_variables(), simplify_rules()

Examples

```
library(validate)
rules <- validator( x >= 0
                  , x <= 0
                  )
detect_fixed_variables(rules)
simplify_fixed_variables(rules)

rules <- validator( x1 + x2 + x3 == 0
                  , x1 + x2 >= 0
                  , x3 >= 0
                  )
simplify_fixed_variables(rules)
```

---

detect_infeasible_rules

Detect which rules cause infeasibility

Description

Detect which rules cause infeasibility. This methods tries to remove the minimum number of rules to make the system mathematically feasible. Note that this may not result in your desired system, because some rules may be more important to you than others. This can be mitigated by supplying weights for the rules. Default weight is 1.

Usage

```
detect_infeasible_rules(x, weight = numeric(), ..., verbose = interactive())
```

Arguments

x	validate::validator() object with rules
weight	optional named numeric() with weights. Unnamed variables in the weight are given the default weight 1.
...	not used
verbose	if TRUE it prints the infeasible rules that have been found.

Value

`character` with the names of the rules that are causing infeasibility.

See Also

Other feasibility: `detect_boundary_cat()`, `detect_boundary_num()`, `detect_contradicting_if_rules()`, `is_contradicted_by()`, `is_infeasible()`, `make_feasible()`

Examples

```
rules <- validator( x > 0)

is_infeasible(rules)

# infeasible system!
rules <- validator( rule1 = x > 0
                  , rule2 = x < 0
                  )

is_infeasible(rules)

detect_infeasible_rules(rules, verbose=TRUE)

# but we want to keep rule1, so specify that it has an Inf weight
detect_infeasible_rules(rules, weight=c(rule1=Inf), verbose=TRUE)

# detect and remove
make_feasible(rules, weight=c(rule1=Inf), verbose = TRUE)

# find out the conflict with rule2
is_contradicted_by(rules, "rule2", verbose = TRUE)
```

---

detect_redundancy *Detect redundant rules*

Description

Detect redundancies in a rule set, but do not remove the rules. A rule in a ruleset can be redundant if it is implied by another rule or by a combination of rules. See the examples for more information.

Usage

```
detect_redundancy(x, ..., verbose = interactive())
```

Arguments

x	`validate::validator()` object with the validation rules.
...	not used.
verbose	if `TRUE` print the redundant rule(s) to the console

Note

For removal of duplicate rules, simplify

See Also

Other redundancy: detect_fixed_variables(), is_implied_by(), remove_redundancy(), simplify_fixed_variable
simplify_rules()

Examples

```
rules <- validator( rule1 = x > 1
                  , rule2 = x > 2
                  )

# rule1 is superfluous
remove_redundancy(rules, verbose=TRUE)

# rule 1 is implied by rule 2
is_implied_by(rules, "rule1", verbose=TRUE)

rules <- validator( rule1 = x > 2
                  , rule2 = x > 2
)

# standout: rule1 and rule2, oldest rules wins
remove_redundancy(rules, verbose=TRUE)

# Note that detection signifies both rules!
detect_redundancy(rules)
```

---

is_categorical *Check whether rules are categorical*

Description

Check whether rules are categorical

Usage

```
is_categorical(x, ...)
```

Arguments

x	validator object
...	not used

Value

logical indicating which rules are purely categorical/logical

Examples

```
v <- validator( A %in% c("a1", "a2")
              , B %in% c("b1", "b2")
              , if (A == "a1") B == "b1"
              , y > x
              )

is_categorical(v)
```

`is_conditional` *Check whether rules are conditional rules*

Description

Check whether rules are conditional rules

Usage

```
is_conditional(rules, ...)
```

Arguments

`rules`	validator object containing validation rules
`...`	not used

Value

logical indicating which rules are conditional

Examples

```
v <- validator( A %in% c("a1", "a2")
              , B %in% c("b1", "b2")
              , if (A == "a1")  x > 1 # conditional
              , if (y > 0) x >= 0 # conditional
              , if (A == "a1") B == "b1" # categorical
              )

is_conditional(v)
```

`is_contradicted_by` *Find out which rules are conflicting*

Description

Find out for a contradicting rule which rules are conflicting. This helps in determining and assessing conflicts in rule sets. Which of the rules should stay and which should go?

Usage

```
is_contradicted_by(x, rule_name, verbose = interactive())
```

Arguments

`x`	`validate::validator()` object with rules.
`rule_name`	`character` with the names of the rules that are causing infeasibility.
`verbose`	if `TRUE` prints the

Value

`character` with conflicting rules.

See Also

Other feasibility: `detect_boundary_cat()`, `detect_boundary_num()`, `detect_contradicting_if_rules()`, `detect_infeasible_rules()`, `is_infeasible()`, `make_feasible()`

Examples

```
rules <- validator( x > 0)

is_infeasible(rules)

# infeasible system!
rules <- validator( rule1 = x > 0
                  , rule2 = x < 0
                  )

is_infeasible(rules)

detect_infeasible_rules(rules, verbose=TRUE)

# but we want to keep rule1, so specify that it has an Inf weight
detect_infeasible_rules(rules, weight=c(rule1=Inf), verbose=TRUE)

# detect and remove
make_feasible(rules, weight=c(rule1=Inf), verbose = TRUE)

# find out the conflict with rule2
is_contradicted_by(rules, "rule2", verbose = TRUE)
```

`is_implied_by` *Find which rule(s) imply a rule*

Description

Find out which rules are causing rule_name(s) to be redundant. A rule set can contain rules that are implied by the other rules, so it is superfluous, see examples.

Usage

```
is_implied_by(x, rule_name, ..., verbose = interactive())
```

Arguments

`x`	`validate::validator()` object with rule
`rule_name`	`character` with the names of the rules to be checked
`...`	not used
`verbose`	if `TRUE` print information to the console

Value

`character` with the names of the rule that cause the implication.

See Also

Other redundancy: `detect_fixed_variables()`, `detect_redundancy()`, `remove_redundancy()`, `simplify_fixed_variables()`, `simplify_rules()`

Examples

```
rules <- validator( rule1 = x > 1
                  , rule2 = x > 2
                  )

# rule1 is superfluous
remove_redundancy(rules, verbose=TRUE)

# rule 1 is implied by rule 2
is_implied_by(rules, "rule1", verbose=TRUE)

rules <- validator( rule1 = x > 2
                  , rule2 = x > 2
)

# standout: rule1 and rule2, oldest rules wins
remove_redundancy(rules, verbose=TRUE)

# Note that detection signifies both rules!
```

```
detect_redundancy(rules)
```

`is_infeasible` *Check the feasibility of a rule set*

Description

An infeasible rule set cannot be satisfied by any data because of internal contradictions: the combination of the rules make it inconsistent. This function checks whether the record-wise linear, categorical and conditional rules in a rule set are consistent. Note that is it wise to also check `detect_contradicting_if_rules()`: conditional If-rules may not be strictly inconsistent, but can be semantically inconsistent.

Usage

```
is_infeasible(x, ..., verbose = interactive())
```

Arguments

`x`	`validator` object with validation rules.
`...`	not used
`verbose`	if `TRUE` print information to the console

Value

TRUE or FALSE

See Also

Other feasibility: `detect_boundary_cat()`, `detect_boundary_num()`, `detect_contradicting_if_rules()`, `detect_infeasible_rules()`, `is_contradicted_by()`, `make_feasible()`

Examples

```
rules <- validator( x > 0)

is_infeasible(rules)

# infeasible system!
rules <- validator( rule1 = x > 0
                  , rule2 = x < 0
                  )

is_infeasible(rules)

detect_infeasible_rules(rules, verbose=TRUE)
```

```
# but we want to keep rule1, so specify that it has an Inf weight
detect_infeasible_rules(rules, weight=c(rule1=Inf), verbose=TRUE)

# detect and remove
make_feasible(rules, weight=c(rule1=Inf), verbose = TRUE)

# find out the conflict with rule2
is_contradicted_by(rules, "rule2", verbose = TRUE)
```

`is_linear` *Check which rules are linear rules.*

Description

Check which rules are linear rules.

Usage

```
is_linear(x, ...)
```

Arguments

`x`	`validate::validator()` object containing data validation rules
`...`	not used

Value

`logical` indicating which rules are (purely) linear.

`make_feasible` *Make an infeasible system feasible.*

Description

Make an infeasible system feasible, by removing the minimum (weighted) number of rules, such that the remaining rules are not conflicting. This function uses `detect_infeasible_rules()` for determining the rules to be removed. Note that this may not result in your desired system, because some rules may be more important. This can be mediated by supplying weights for the rules. Default weight is 1.

Usage

```
make_feasible(x, ..., verbose = interactive())
```

Arguments

x	validate::validator() object with the validation rules.
...	passed to detect_infeasible_rules()
verbose	if TRUE print information to the console

Details

Also make_feasible() does not check for contradictions in if rules, so it is wise to also check detect_contradicting_if_rules() after making the system feasible.

Value

validate::validator() object with feasible rules.

See Also

Other feasibility: detect_boundary_cat(), detect_boundary_num(), detect_contradicting_if_rules(), detect_infeasible_rules(), is_contradicted_by(), is_infeasible()

Examples

```
rules <- validator( x > 0)

is_infeasible(rules)

# infeasible system!
rules <- validator( rule1 = x > 0
                  , rule2 = x < 0
                  )

is_infeasible(rules)

detect_infeasible_rules(rules, verbose=TRUE)

# but we want to keep rule1, so specify that it has an Inf weight
detect_infeasible_rules(rules, weight=c(rule1=Inf), verbose=TRUE)

# detect and remove
make_feasible(rules, weight=c(rule1=Inf), verbose = TRUE)

# find out the conflict with rule2
is_contradicted_by(rules, "rule2", verbose = TRUE)
```

`remove_redundancy` *Remove redundant rules*

Description

Simplify a rule set by removing redundant rules, i.e. rules that are implied by other rules.

Usage

```
remove_redundancy(x, ..., verbose = interactive())
```

Arguments

`x`	`validate::validator()` object with validation rules.
`...`	not used
`verbose`	if `TRUE` print the remove rules to the console.

Value

simplified `validate::validator()` object, in which redundant rules are removed.

See Also

Other redundancy: `detect_fixed_variables()`, `detect_redundancy()`, `is_implied_by()`, `simplify_fixed_variable`
`simplify_rules()`

Examples

```
rules <- validator( rule1 = x > 1
                  , rule2 = x > 2
                  )

# rule1 is superfluous
remove_redundancy(rules, verbose=TRUE)

# rule 1 is implied by rule 2
is_implied_by(rules, "rule1", verbose=TRUE)

rules <- validator( rule1 = x > 2
                  , rule2 = x > 2
)

# standout: rule1 and rule2, oldest rules wins
remove_redundancy(rules, verbose=TRUE)

# Note that detection signifies both rules!
detect_redundancy(rules)
```

`simplify_conditional` *Simplify conditional statements*

Description

Conditional rules (if-rules) may be constrained by the others rules in a validation rule set. This procedure tries to simplify conditional statements.

Usage

```
simplify_conditional(x, ...)
```

Arguments

x	`validate::validator()` object with the validation rules.
...	not used.

Value

`validate::validator()` simplified rule set.

References

TODO non-constraining, non-relaxing

Examples

```
library(validate)

# superfluous conditions
rules <- validator(
  r1 = if (x > 0) x < 1,
  r2 = if (y > 0 && y > 1) z > 0
)
# r1: x > 0 is superfluous because r1 equals (x <= 0) | (x < 1)
# r2: y > 0 is superfluous because r2 equals (y <= 0) | (y <= 1) | (z > 0)
simplify_conditional(rules)

# non-relaxing clause
rules <- validator( r1 = if (x > 1) y > 3
                  , r2 = y < 2
                  )
# y > 3 is always FALSE so r1 can be simplified
simplify_conditional(rules)

# non-constraining clause
rules <- validator( r1 = if (x > 0) y > 0
                  , r2 = if (x < 1) y > 1
                  )
```

```
simplify_conditional(rules)

rules <- validator( r1 = if (A == "a1") x > 0
                  , r2 = if (A == "a2") x > 1
                  , r3 = A == "a1"
                  )
simplify_conditional(rules)
```

`simplify_fixed_variables` *Simplify fixed variables*

Description

Detect variables of which the values are restricted to a single value by the rule set. Simplify the rule set by replacing fixed variables with these values.

Usage

```
simplify_fixed_variables(x, eps = 1e-08, ...)
```

Arguments

`x`	`validate::validator()` object with validation rules
`eps`	detected fixed values will have this precission.
`...`	passed to `substitute_values()`.

Value

`validate::validator()` object in which

See Also

Other redundancy: `detect_fixed_variables()`, `detect_redundancy()`, `is_implied_by()`, `remove_redundancy()`, `simplify_rules()`

Examples

```
library(validate)
rules <- validator( x >= 0
                  , x <= 0
                  )
detect_fixed_variables(rules)
simplify_fixed_variables(rules)

rules <- validator( x1 + x2 + x3 == 0
                  , x1 + x2 >= 0
                  , x3 >= 0
                  )
simplify_fixed_variables(rules)
```

`simplify_rules` *Simplify a rule set*

Description

Simplifies a rule set set by applying different simplification methods. This is a convenience function that works in common cases. The following simplification methods are executed:

- `substitute_values()`: filling in any parameters that are supplied via `.values` or `....`
- `simplify_fixed_variables()`: find out if there are fixed values. If this is the case, they are substituted.
- `simplify_conditional()`: Simplify conditional statements, by removing clauses that are superfluous.
- `remove_redundancy()`: remove redundant rules.

For more control, these methods can be called separately.

Usage

```
simplify_rules(.x, .values = list(...), ...)
```

Arguments

`.x`	`validate::validator()` object with the rules to be simplified.
`.values`	optional named list with values that will be substituted.
`...`	parameters that will be used to substitute values.

Details

Note that it is wise to call `detect_contradicting_if_rules()` before calling this function.

See Also

Other redundancy: `detect_fixed_variables()`, `detect_redundancy()`, `is_implied_by()`, `remove_redundancy()`, `simplify_fixed_variables()`

Examples

```
rules <- validator( x > 0
                  , if (x > 0) y == 1
                  , A %in% c("a1", "a2")
                  , if (A == "a1") y > 1
                  )

simplify_rules(rules)
```

`substitute_values` *substitute a value in a rule set*

Description

Substitute values / fill in a value for one ore more variables, thereby simplifying the rule set. Rules that evaluate to TRUE because of the substitution are removed.

Usage

```
substitute_values(.x, .values = list(...), ..., .add_constraints = TRUE)
```

Arguments

`.x`	`validator` object with rules
`.values`	(optional) named list with values for variables to substitute
`...`	alternative way of supplying values for variables (see examples).
`.add_constraints`	`logical`, should values be added as constraints to the resulting validator object?

Examples

```
library(validate)
rules <- validator( rule1 = z > 1
                  , rule2 = y > z
                  )
# rule1 is dropped, since it always is true
substitute_values(rules, list(z=2))

# you can also supply the values as separate parameters
substitute_values(rules, z = 2)

# you can choose to not add substituted values as a constraint
substitute_values(rules, z = 2, .add_constraints = FALSE)

rules <- validator( rule1 = if (gender == "male") age >= 18 )
substitute_values(rules, gender="male")
substitute_values(rules, gender="female")
```

validatetools *Tools for validation rules*

Description

validatetools is a utility package for managing validation rule sets that are defined with validate::validate(). In production systems validation rule sets tend to grow organically and accumulate redundant or (partially) contradictory rules. validatetools helps to identify problems with large rule sets and includes simplification methods for resolving issues.

Problem detection

The following methods allow for problem detection:

- is_infeasible() checks a rule set for feasibility, i.e. if it contains contradicting rules making it infeasible. An infeasible system must be corrected to be useful, because it means that no record can satisfy all rules.
- detect_infeasible_rules() detects which rules cause infeasibility.
- detect_contradicting_if_rules() detects which if rules are conflicting.
- detect_boundary_num() shows for each numerical variable the allowed range of values.
- detect_boundary_cat() shows for each categorical variable the allowed range of values.
- detect_fixed_variables() shows variables whose value is fixated by the rule set.
- detect_redundancy() shows which rules are already implied by other rules.

Simplifying rule set

The following methods detect possible simplifications and apply them to a rule set.

- substitute_values(): replace variables with constants.
- simplify_fixed_variables(): substitute the fixed variables with their values in a rule set.
- simplify_conditional(): remove redundant (parts of) conditional rules.
- remove_redundancy(): remove redundant rules.

Author(s)

Maintainer: Edwin de Jonge <edwindjonge@gmail.com> (ORCID)

Authors:

- Mark van der Loo <mark.vanderloo@gmail.com>

Other contributors:

- Jacco Daalmans [contributor]

References

Statistical Data Cleaning with Applications in R, Mark van der Loo and Edwin de Jonge, ISBN: 978-1-118-89715-7

See Also

Useful links:

- https://github.com/data-cleaning/validatetools
- Report bugs at https://github.com/data-cleaning/validatetools/issues

Index

∗ **feasibility**
  detect_boundary_cat, 2
  detect_boundary_num, 3
  detect_contradicting_if_rules, 4
  detect_infeasible_rules, 6
  is_contradicted_by, 10
  is_infeasible, 12
  make_feasible, 13
∗ **redundancy**
  detect_fixed_variables, 5
  detect_redundancy, 7
  is_implied_by, 11
  remove_redundancy, 15
  simplify_fixed_variables, 17
  simplify_rules, 18

data.frame(), *3*
detect_boundary_cat, 2, *4, 5*, *7*, *10*, *12*, *14*
detect_boundary_cat(), *20*
detect_boundary_num, *2*, 3, *5*, *7*, *10*, *12*, *14*
detect_boundary_num(), *20*
detect_contradicting_if_rules, *2*, *4*, 4, *7*, *10*, *12*, *14*
detect_contradicting_if_rules(), *18*, *20*
detect_fixed_variables, 5, *8*, *11*, *15*, *17, 18*
detect_fixed_variables(), *4*, *20*
detect_infeasible_rules, *2*, *4, 5*, 6, *10*, *12*, *14*
detect_infeasible_rules(), *13, 14*, *20*
detect_redundancy, *6*, 7, *11*, *15*, *17, 18*
detect_redundancy(), *20*

is_categorical, 8
is_conditional, 9
is_contradicted_by, *2*, *4, 5*, *7*, 10, *12*, *14*
is_implied_by, *6*, *8*, 11, *15*, *17, 18*
is_infeasible, *2*, *4, 5*, *7*, *10*, 12, *14*
is_infeasible(), *20*
is_linear, 13

make_feasible, *2*, *4, 5*, *7*, *10*, *12*, 13

numeric(), *6*

remove_redundancy, *6*, *8*, *11*, 15, *17, 18*
remove_redundancy(), *18*, *20*

simplify_conditional, 16
simplify_conditional(), *18*, *20*
simplify_fixed_variables, *6*, *8*, *11*, *15*, 17, *18*
simplify_fixed_variables(), *6*, *18*, *20*
simplify_rules, *6*, *8*, *11*, *15*, *17*, 18
substitute_values, 19
substitute_values(), *17, 18*, *20*

validate::validate(), *20*
validate::validator(), *2, 3*, *6, 7*, *10, 11*, *13–18*
validatetools, 20
validatetools-package *(*validatetools*)*, 20