Package: gradethis (via r-universe)

May 16, 2026

Type Package

Title Automated Feedback for Student Exercises in 'learnr' Tutorials

Version 0.2.14

Description Pairing with the 'learnr' R package, 'gradethis' provides multiple methods to grade 'learnr' exercises. To learn more about 'learnr' tutorials, please visit <https://rstudio.github.io/learnr/>.

License MIT + file LICENSE

URL https://pkgs.rstudio.com/gradethis/, https://rstudio.github.io/learnr/, https://github.com/rstudio/gradethis

BugReports https://github.com/rstudio/gradethis/issues

Depends R (>= 3.2.0)

Imports checkmate, commonmark, ellipsis, glue, htmltools, learnr (>= 0.10.1.9008), lifecycle, magrittr, purrr, rlang, rstudioapi, utils, waldo, withr

Suggests DBI, ggcheck (>= 0.0.5), ggplot2, knitr, rmarkdown, spelling, testthat (>= 3.0.0)

VignetteBuilder knitr

Remotes rstudio/ggcheck, rstudio/learnr

Config/Needs/connect rsconnect

Config/Needs/coverage covr

Config/Needs/website pkgdown, tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

Language en-US

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Config/pak/sysreqs cmake make libuv1-dev zlib1g-dev

Repository https://rstudio.r-universe.dev

Date/Publication 2023-07-05 17:03:51 UTC

RemoteUrl https://github.com/rstudio/gradethis

RemoteRef HEAD

RemoteSha 502af700437db1b648a35200ce7b940db0cc3d45

Contents

`code_feedback` *Provide automated code feedback*

Description

Generate a message describing the first instance of a code mismatch. Three functions are provided for working with code feedback: `code_feedback()` does the comparison and returns a character description of the mismatch, or a `NULL` if no differences are found. `maybe_code_feedback()` is designed to be used inside `fail()` and related `graded()` messages, as in `"{maybe_code_feedback()}"`. And `give_code_feedback()` gives you a way to add code feedback to any `fail()` message in a `grade_this()` or `grade_result()` checking function.

Usage

```
code_feedback(
  user_code = .user_code,
  solution_code = .solution_code_all,
  user_env = .envir_result,
  solution_env = .envir_solution,
  ...,
  allow_partial_matching = getOption("gradethis.allow_partial_matching", TRUE)
)

maybe_code_feedback(
  user_code = get0(".user_code", parent.frame()),
  solution_code = get0(".solution_code_all", parent.frame()),
  user_env = get0(".envir_result", parent.frame(), ifnotfound = parent.frame()),
  solution_env = get0(".envir_solution", parent.frame(), ifnotfound = parent.frame()),
  ...,
  allow_partial_matching = getOption("gradethis.allow_partial_matching", TRUE),
  default = "",
  before = getOption("gradethis.maybe_code_feedback.before", " "),
  after = getOption("gradethis.maybe_code_feedback.after", NULL),
  space_before = deprecated(),
  space_after = deprecated()
)

give_code_feedback(
  expr,
  ...,
  env = parent.frame(),
  location = c("after", "before")
)
```

Arguments

user_code, solution_code	String containing user or solution code. By default, when used in grade_this(), .user_code is retrieved for the .user_code. solution_code may also be a list containing multiple solution variations, so by default in grade_this() .solution_code_all is found and used for solution_code. You may also use .solution_code if there is only one solution.
user_env	Environment used to standardize formals of the user code. Defaults to retrieving .envir_result from the calling environment. If not found, the parent.frame() will be used.
solution_env	Environment used to standardize formals of the solution code. Defaults to retrieving .envir_solution from the calling environment. If not found, the parent.frame() will be used.
...	Ignored in code_feedback() and maybe_code_feedback(). In give_code_feedback(), ... are passed to maybe_code_feedback().

`allow_partial_matching`
: A logical. If `FALSE`, the partial matching of argument names is not allowed and e.g. `runif(1, mi = 0)` will return a message indicating that the full formal name `min` should be used. The default is set via the `gradethis.allow_partial_matching` option, or by `gradethis_setup()`.

`default`
: Default value to return if no code feedback is found or code feedback can be provided.

`before, after`
: Strings to be added before or after the code feedback message to ensure the message is properly formatted in your feedback.

`space_before, space_after`
: Deprecated. Use `before` and `after`.

`expr`
: A grading function — like `grade_this()` or `grade_result()` — or a character string. The code feedback will be appended to the message of any incorrect grades using `maybe_code_feedback()`, set to always include the code feedback, if possible. If `expr` is a character string, `"{maybe_code_feedback()}"` is pasted into the string, without customization.

`env`
: Environment used to standardize formals of the user and solution code. Defaults to retrieving .envir_result and .envir_solution from `parent.frame()`.

`location`
: Should the code feedback message be added before or after?

Value

- `code_feedback()` returns a character value describing the difference between the student's submitted code and the solution. If no discrepancies are found, `code_feedback()` returns `NULL`.
- `maybe_code_feedback()` always returns a string for safe use in glue strings. If no discrepancies are found, it returns an empty string.
- `give_code_feedback()` catches `fail()` grades and adds code feedback to the feedback message using `maybe_code_feedback()`.

Functions

- `code_feedback()`: Determine code feedback by comparing the user's code to the solution.
- `maybe_code_feedback()`: Return `code_feedback()` result when possible. Useful when setting default `fail()` glue messages. For example, if there is no solution, no code feedback will be given.
- `give_code_feedback()`: Appends `maybe_code_feedback()` to the message generated by incorrect grades.

Code differences

There are many different ways that code can be different, yet still the same. Here is how we detect code differences:

1. If the single values are different. Ex: `log(2)` vs `log(3)`
2. If the function call is different. Ex: `log(2)` vs `sqrt(2)`

3. Validate the user code can be standardized via `rlang::call_standardise()`. The `env` parameter is important for this step as **gradethis** does not readily know about user defined functions. Ex: `read.csv("file.csv")` turns into `read.csv(file = "file.csv")`
4. If multiple formals are matched. Ex: `read.csv(f = "file.csv")` has `f` match to `file` and `fill`.
5. Verify that every named argument in the solution appears in the user code. Ex: If the solution is `read.csv("file.csv", header = TRUE)`, `header` must exist.
6. Verify that the user did not supply extra named arguments to `...`. Ex: `mean(x = 1:10, na.rm = TRUE)` vs `mean(x = 1:10)`
7. Verify that every named argument in the solution matches the value of the corresponding user argument. Ex: `read.csv("file.csv", header = TRUE)` vs `read.csv("file.csv", header = FALSE)`
8. Verify that the remaining arguments of the user and solution code match in order and value. Ex: `mean(1:10, 0.1)` vs `mean(1:10, 0.2)`

Examples

```
# code_feedback() ------------------------------------------------------

# Values are same, so no differences found
code_feedback("log(2)", "log(2)")

# Functions are different
code_feedback("log(2)", "sqrt(2)")

# Standardize argument names (no differences)
code_feedback("read.csv('file.csv')", "read.csv(file = 'file.csv')")

# Partial matching is not allowed
code_feedback("read.csv(f = 'file.csv')", "read.csv(file = 'file.csv')")

# Feedback will spot differences in argument values...
code_feedback(
  "read.csv('file.csv', header = FALSE)",
  "read.csv('file.csv', header = TRUE)"
)

# ...or when arguments are expected to appear in a call...
code_feedback("mean(1:10)", "mean(1:10, na.rm = TRUE)")

# ...even when the expected argument matches the function's default value
code_feedback("read.csv('file.csv')", "read.csv('file.csv', header = TRUE)")

# Unstandardized arguments will match by order and value
code_feedback("mean(1:10, 0.1)", "mean(1:10, 0.2)")


# give_code_feedback() -------------------------------------------------

# We'll use this example of an incorrect exercise submission throughout
```

```
submission_wrong <- mock_this_exercise(
  .user_code = "log(4)",
  .solution_code = "sqrt(4)"
)

# To add feedback to *any* incorrect grade,
# wrap the entire `grade_this()` call in `give_code_feedback()`:
grader <-
  # ```{r example-check}
  give_code_feedback(grade_this({
    pass_if_equal(.solution, "Good job!")
    if (.result < 2) {
      fail("Too low!")
    }
    fail()
  }))
# ```
grader(submission_wrong)

# Or you can wrap the message of any fail() directly:
grader <-
  # ```{r example-check}
  grade_this({
    pass_if_equal(.solution, "Good job!")
    if (.result < 2) {
      fail(give_code_feedback("Too low!"))
    }
    fail()
  })
# ```
grader(submission_wrong)

# Typically, grade_result() doesn't include code feedback
grader <-
  # ```{r example-check}
  grade_result(
    fail_if(~ round(.result, 0) != 2, "Not quite!")
  )
# ```
grader(submission_wrong)

# But you can use give_code_feedback() to append code feedback
grader <-
  # ```{r example-check}
  give_code_feedback(grade_result(
    fail_if(~ round(.result, 0) != 2, "Not quite!")
  ))
# ```
grader(submission_wrong)

# The default `grade_this_code()` `incorrect` message always adds code feedback,
# so be sure to remove \"{maybe_code_feedback()}\" from the incorrect message
grader <-
```

```
  # ```{r example-check}
  give_code_feedback(grade_this_code(incorrect = "{random_encouragement()}"))
# ```
grader(submission_wrong)
```

---

debug_this *Debug an exercise submission*

Description

When used in a *-check chunk or inside grade_this(), debug_this() displays in the **learnr** tutorial a complete listing of the variables and environment available for checking. This can be helpful when you need to debug an exercise and a submission.

Usage

```
debug_this(check_env = parent.frame())
```

Arguments

check_env	A grade checking environment. You can use mock_this_exercise() to prepare a mocked exercise submission environment. Otherwise, you don't need to use or set this argument.

Value

Returns a neutral grade containing a message that includes any and all information available about the exercise and the current submission. The output lets you visually explore the objects available for use within your grade_this() grading code.

Debugging exercises

debug_this() gives you a few ways to see the objects that are available inside grade_this() for you to use when grading exercise submissions. Suppose we have this example exercise:

````
```{r example-setup}
x <- 1
```

```{r example, exercise = TRUE}
# user submits
y <- 2
x + y
```

```{r example-solution}
x + 3
```
````

Always debug:
The first method is the most straight-forward. Inside the `*-check` or `*-error-check` chunks for your exercise, simply call `debug_this()`:

````
```{r example-check}
debug_this()
```
````

Every time you submit code for feedback via **Submit Answer**, the debug information will be printed.

Debug specific cases:
On the other hand, if you want to debug a specific submission, such as a case where a submission isn't matching any of your current grading conditions, you can call `debug_this()` wherever you like inside `grade_this()`.

````
```{r example-check}
grade_this({
  pass_if_equal(3, "Good work?")

  # debug the submission if it is somehow equal to 2
  if (.result == 2) {
    debug_this()
  }
})
```
````

Debug default fail condition:
It's common to have the grade-checking code default to an incorrect grade with code feedback by calling `fail()` at the end of the checking code in `grade_this()`. During development of a tutorial, you may want to have this default `fail()` return the debugging information rather than a failure.
By setting the global option `gradethis.fail` to use `debug_this()`,

````
```{r setup}
library(learnr)
library(gradethis)
gradethis_setup()

option(gradethis.fail = "{debug_this()}")
```
````

you can see the values that are available to you during the submission check whenever your test submissions pass through your other checks.

````
```{r example-check}
grade_this({
  pass_if_equal(3, "Good work?")

  fail()
})
```
````

Don't forget to reset or unset the `gradethis.fail` option when you're done working on your tutorial.

Examples

```
# Suppose we have an exercise (guess the number 42). Mock a submission:
submission <- mock_this_exercise(.user_code = 40, .solution_code = 11 + 31)

# Call `debug_this()` inside your *-check chunk, is equivalent to
debug_this()(submission)$message

# The remaining examples produce equivalent output
## Not run:
# Or you can call `debug_this()` inside a `grade_this()` call
# at the point where you want to get debug feedback.
grade_this({
  pass_if_equal(42, "Good stuff!")

  # Find out why this is failing??
  debug_this()
})(submission)

# Set default `fail()` message to show debug information
# (for tutorial development only!)
old_opts <- options(gradethis.fail = "{debug_this()}")

grade_this({
  pass_if_equal(42, "Good stuff!")

  fail()
})(submission)

# default fail() will show debug until you reset gradethis.fail option
options(old_opts)

## End(Not run)
```

`fail_if_code_feedback` *Signal a failing grade if mistakes are detected in the submitted code*

Description

`fail_if_code_feedback()` uses `code_feedback()` to detect if there are differences between the user's submitted code and the solution code (if available). If the exercise does not have an associated solution, or if there are no detected differences between the user's and the solution code, no grade is returned.

See `graded()` for more information on **gradethis** grade-signaling functions.

Usage

```
fail_if_code_feedback(
  message = NULL,
  user_code = .user_code,
  solution_code = .solution_code_all,
  ...,
  env = parent.frame(),
  hint = TRUE,
  encourage = getOption("gradethis.fail.encourage", FALSE),
  allow_partial_matching = getOption("gradethis.allow_partial_matching", TRUE)
)
```

Arguments

message	A character string of the message to be displayed. In all grading helper functions other than graded(), message is a template string that will be processed with glue::glue().
user_code, solution_code	String containing user or solution code. By default, when used in grade_this(), .user_code is retrieved for the .user_code. solution_code may also be a list containing multiple solution variations, so by default in grade_this() .solution_code_all is found and used for solution_code. You may also use .solution_code if there is only one solution.
...	Arguments passed on to graded correct A logical value of whether or not the checked code is correct. type,location The type and location of the feedback object provided to **learnr**. See https://rstudio.github.io/learnr/exercises.html#Custom_checking for more details. type may be one of "auto", "success", "info", "warning", "error", or "custom". location may be one of "append", "prepend", or "replace".
env	Environment used to standardize formals of the user and solution code. Defaults to retrieving .envir_result and .envir_solution from parent.frame().
hint	Include a code feedback hint with the failing message? This argument only applies to fail() and fail_if_equal() and the message is added using the default options of give_code_feedback() and maybe_code_feedback(). The default value of hint can be set using gradethis_setup() or the gradethis.fail.hint option.
encourage	Include a random encouraging phrase with random_encouragement()? The default value of encourage can be set using gradethis_setup() or the gradethis.fail.encourage option.
allow_partial_matching	A logical. If FALSE, the partial matching of argument names is not allowed and e.g. runif(1, mi = 0) will return a message indicating that the full formal name min should be used. The default is set via the gradethis.allow_partial_matching option, or by gradethis_setup().

Value

Signals an incorrect grade with feedback if there are differences between the submitted user code and the solution code. If solution code is not available, no grade is returned.

See Also

Other grading helper functions: graded(), pass(), fail(), pass_if(), fail_if(), pass_if_equal(), fail_if_equal().

Examples

```
# Suppose the exercise prompt is to generate 5 random numbers, sampled from
# a uniform distribution between 0 and 1. In this exercise, you know that
# you shouldn't have values outside of the range of 0 or 1, but you'll
# otherwise need to check the submitted code to know that the student has
# chosen the correct sampling function.

grader <-
  # ```{r example-check}
  grade_this({
    fail_if(length(.result) != 5, "I expected 5 numbers.")
    fail_if(
      any(.result < 0 | .result > 1),
      "I expected all numbers to be between 0 and 1."
    )

    # Specific checks passed, but now we want to check the code.
    fail_if_code_feedback()

    # All good!
    pass()
  })
# ```

.solution_code <- "
# ```{r example-check}
  runif(5)
# ```
"


# Not 5 numbers...
grader(mock_this_exercise(runif(1), !!.solution_code))

# Not within [0, 1]...
grader(mock_this_exercise(rnorm(5), !!.solution_code))

# Passes specific checks, but hard to tell so check the code...
grader(mock_this_exercise(runif(5, 0.25, 0.75), !!.solution_code))
grader(mock_this_exercise(rbinom(5, 1, 0.5), !!.solution_code))

# Perfect!
grader(mock_this_exercise(runif(n = 5), !!.solution_code))
```

`fail_if_error` *Fail if grading code produces an error*

Description

When grading code involves unit-style testing, you may want to use **testthat** expectation function to test the user's submitted code. In these cases, to differentiate between expected errors and internal errors indicative of issues with the grading code, **gradethis** requires that authors wrap assertion-style tests in `fail_if_error()`. This function catches any errors and converts them into `fail()` grades. It also makes the error and its message available for use in the `message` glue string as `.error` and `.error_message` respectively.

Usage

```
fail_if_error(
  expr,
  message = "{.error_message}",
  ...,
  env = parent.frame(),
  hint = TRUE,
  encourage = getOption("gradethis.fail.encourage", FALSE)
)
```

Arguments

`expr`	An expression to evaluate that whose errors are safe to be converted into failing grades with `fail()`.
`message`	A glue string containing the feedback message to be returned to the user. Additional `.error` and `.error_message` objects are made available for use in the message.
`...`	Additional arguments passed to `graded()` or additional data to be included in the feedback object.
`env`	environment to evaluate the glue `message`. Most users of **gradethis** will not need to use this argument.
`hint`	Include a code feedback hint with the failing message? This argument only applies to `fail()` and `fail_if_equal()` and the message is added using the default options of `give_code_feedback()` and `maybe_code_feedback()`. The default value of `hint` can be set using `gradethis_setup()` or the `gradethis.fail.hint` option.
`encourage`	Include a random encouraging phrase with `random_encouragement()`? The default value of `encourage` can be set using `gradethis_setup()` or the `gradethis.fail.encourage` option.

Value

If an error occurs while evaluating `expr`, the error is returned as a `fail()` grade. Otherwise, no value is returned.

See Also

Other grading helper functions: `graded()`, `pass()`, `fail()`, `pass_if()`, `fail_if()`, `pass_if_equal()`, `fail_if_equal()`.

Examples

```
# The user is asked to add 2 + 2, but they take a shortcut
ex <- mock_this_exercise("'4'")

# Normally, grading code with an author error returns an internal problem grade
grade_author_mistake <- grade_this({
  if (identical(4)) {
    pass("Great work!")
  }
  fail()
})(ex)

# This returns a "problem occurred" grade
grade_author_mistake
# ...that also includes information about the error (not shown to users)
grade_author_mistake$error

# But sometimes we'll want to use unit-testing helper functions where we know
# that an error is indicative of a problem in the users' code
grade_this({
  fail_if_error({
    testthat::expect_length(.result, 1)
    testthat::expect_true(is.numeric(.result))
    testthat::expect_equal(.result, 4)
  })
  pass("Good job!")
})(ex)

# Note that you don't need to reveal the error message to the user
grade_this({
  fail_if_error(
    message = "Your result isn't a single numeric value.",
    {
      testthat::expect_length(.result, 1)
      testthat::expect_true(is.numeric(.result))
      testthat::expect_equal(.result, 4)
    }
  )
  pass("Good job!")
})(ex)
```

grade_this *Grade a student's submission using custom logic*

Description

`grade_this()` allows instructors to write custom logic to evaluate, grade and give feedback to students. To use `grade_this()`, call it directly in your `*-check` chunk:

````
```{r example-check}
grade_this({
  # custom checking code appears here
  if (identical(.result, .solution)) {
    pass("Great work!")
  }
  fail("Try again!")
})
```
````

`grade_this()` makes available a number of objects based on the exercise and the student's submission that can be used to evaluate the student's submitted code. See `?"grade_this-objects"` for more information about these objects.

As the instructor, you are free to use any logic to determine a student's grade as long as a `graded()` object is signaled. The check code can also contain **testthat** expectation code. Failed **testthat** expectations will be turned into `fail()`ed grades with the corresponding message.

A final grade is signaled from `grade_this()` using the `graded()` helper functions, which include `pass()`, `fail()`, among others. `grade_this()` uses condition handling to short-circuit further evaluation when a grade is reached. This means that you may also signal a failing grade using any of the `expect_*()` functions from **testthat**, other functions designed to work with **testthat**, such as **checkmate**, or standard R errors via `stop()`. Learn more about this behavior in `graded()` in the section **Return a grade immediately**.

Usage

```
grade_this(
  expr,
  ...,
  maybe_code_feedback = getOption("gradethis.maybe_code_feedback", TRUE)
)
```

Arguments

`expr`	The grade-checking expression to be evaluated. This expression must either signal a grade via `pass()` or `fail()` functions or their sibling functions. By default, errors in this expression are converted to "internal problem" grades that mask the error for the user. If your grading logic relies on unit-test-styled functions, such as those from **testthat**, you can use `fail_if_error()` to convert errors into `fail()` grades.

`...` Ignored

`maybe_code_feedback`
: Should `maybe_code_feedback()` provide code feedback when used in a `graded()` message? The default value can be set with `gradethis_setup()`.

  Typically, `maybe_code_feedback()` is called in the default `fail()` message (the default can be customized the `fail` argument of `gradethis_setup()`). If the `maybe_code_feedback` argument is `FALSE`, `maybe_code_feedback()` returns an empty string.

Value

Returns a function whose first parameter will be an environment containing objects specific to the exercise and submission (see **Available variables**). For local testing, you can create a version of the expected environment for a mock exercise submission with `mock_this_exercise()`. Calling the returned function on the exercise-checking environment will evaluate the grade-checking `expr` and return a final grade via `graded()`.

See Also

`grade_this_code()`, `mock_this_exercise()`, `gradethis_demo()`

Examples

```
# For an interactive example run: gradethis_demo()

# Suppose we have an exercise that prompts students to calculate the
# average height of Loblolly pine trees using the `Loblolly` data set.
# We might write an exercise `-check` chunk like the one below.
#
# Since grade_this() returns a function, we'll save the result of this
# "chunk" as `grader()`, which can be called on an exercise submission
# to evaluate the student's code, which we'll simulate with
# `mock_this_exercise()`.

grader <-
  # ```{r example-check}
  grade_this({
    if (length(.result) != 1) {
      fail("I expected a single value instead of {length(.result)} values.")
    }

    if (is.na(.result)) {
      fail("I expected a number, but your code returned a missing value.")
    }

    avg_height <- mean(Loblolly$height)
    if (identical(.result, avg_height)) {
      pass("Great work! The average height is {round(avg_height, 2)}.")
    }

    # Always end grade_this() with a default grade.
```

```
    # By default fail() will also give code feedback,
    # if a solution is available.
    fail()
  })
# ```

# Simulate an incorrect answer: too many values...
grader(mock_this_exercise(.user_code = Loblolly$height[1:2]))

# This student submission returns a missing value...
grader(mock_this_exercise(mean(Loblolly$Seed)))
# This student submission isn't caught by any specific tests,
# the final grade is determined by the default (last) value in grade_this()
grader(mock_this_exercise(mean(Loblolly$age)))

# If you have a *-solution chunk,
# fail() without arguments gives code feedback...
grader(
  mock_this_exercise(
    .user_code = mean(Loblolly$age),
    .solution_code = mean(Loblolly$height)
  )
)

# Finally, the "student" gets the correct answer!
grader(mock_this_exercise(mean(Loblolly$height)))
```

`grade_this-objects` *Checking environment objects for use in* `grade_this()`

Description

`grade_this()` allows instructors to determine a grade and to create custom feedback messages using custom R code. To facilitate evaluating the exercise, `grade_this()` makes available a number of objects that can be referenced within the `{ ... }` expression.

All of the objects provided by `learnr` to an exercise checking function are available for inspection. To avoid name collisions with user or instructor code, the names of these objects all start with `..`

- `.label`: The exercise label.
- `.engine`: The exercise engine, typically 'r'.
- `.last_value`: The last value returned from evaluating the user's exercise submission.
- `.solution_code`: A string containing the code provided within the `*-solution` chunk for the exercise.
- `.user_code`: A string containing the code submitted by the user.
- `.check_code`: A string containing the code provided within the `*-check` or `*-code-check` chunk for the exercise.

- `.envir_prep`: A copy of the R environment after running the exercise setup code and before the execution of the student's submitted code.
- `.envir_result`: The R environment after running the student's submitted code.
- `.envir_solution`: The R environment after running the solution code.
- `.evaluate_result`: The return value from the `evaluate::evaluate()` function (see learnr's documentation).
- `.stage`: The current checking stage in the learnr exercise evaluation lifecycle: 'code_check', 'error_check', or 'check'

In addition, **gradethis** has provided some extra objects:

- `.user`, `.result`: The last value returned from evaluating the user's exercise submission.
- `.solution`: The last value returned from evaluating the `.solution_code` for the exercise (evaluated in `.envir_prep`).
- `.solution_all`: A list containing all solutions when multiple solutions are provided in the `*-solution` chunk for the exercise. Solutions are separated by header comments, e.g. `# base_r ----`.
- `.solution_code_all`: A list containing the code of all solutions when multiple solutions are provided in the `*-solution` chunk for the exercise. Solutions are separated by header comments, e.g. `# base_r ----`.

Usage

```
.result

.user

.last_value

.solution

.solution_all

.user_code

.solution_code

.solution_code_all

.envir_prep

.envir_result

.envir_solution

.evaluate_result

.label
```

`.stage`

`.engine`

Format

An object of class `.result` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.user` (inherits from `.result`, `gradethis_placeholder`) of length 0.

An object of class `.last_value` (inherits from `.result`, `gradethis_placeholder`) of length 0.

An object of class `.solution` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.solution_all` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.user_code` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.solution_code` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.solution_code_all` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.envir_prep` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.envir_result` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.envir_solution` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.evaluate_result` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.label` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.stage` (inherits from `gradethis_placeholder`) of length 0.

An object of class `.engine` (inherits from `gradethis_placeholder`) of length 0.

---

grade_this_code *Grade student code against a solution*

Description

`grade_this_code()` compares student code to a solution (i.e. model code) and describes the first way in which the student code differs. If the student code exactly matches the solution, `grade_this_code()` returns a customizable success message (`correct`). If the student code does not match the solution, a customizable incorrect message (`incorrect`) can also be provided.

In most cases, to use `grade_this_code()`, ensure that your exercise has a `-solution` chunk:

````
```{r example-solution}
sqrt(log(1))
```
````

Then, call `grade_this_code()` in your exercise's `-check` or `-code-check` chunk:

````
```{r example-check}
grade_this_code()
```
````

If `grade_this_code()` is called in a `-code-check` chunk and returns feedback, either passing or failing feedback, then the user's code is not executed. If you want the user to see the output of their code, call `grade_this_code()` in the `-check` chunk. You can also use `grade_this_code()` as a pre-check to avoid running code when it fails or passes by calling `grade_this_code()` inside the `-code-check` chunk and setting `action = "pass"` or `action = "fail"` to only return feedback when the user's code passes or fails, respectively. (Note: requires **learnr** version 0.10.1.9017 or later.)

Learn more about how to use `grade_this_code()` in the **Details** section below.

Usage

```
grade_this_code(
  correct = getOption("gradethis.code_correct", getOption("gradethis.pass", "Correct!")),
  incorrect = getOption("gradethis.code_incorrect", getOption("gradethis.fail",
    "Incorrect")),
  ...,
  allow_partial_matching = getOption("gradethis.allow_partial_matching", TRUE),
  action = c("both", "pass", "fail")
)
```

Arguments

`correct`	A glue-able character string to display if the student answer matches a known correct answer.
`incorrect`	A glue-able character string to display if the student answer does not match the known correct answer. Use `code_feedback()` in this string to control the placement of the auto-generated feedback message produced by comparing the student's submission with the solution. Use a string that doesn't include `code_feedback()` to grade the student's code without providing feedback.
`...`	Ignored
`allow_partial_matching`	A logical. If `FALSE`, the partial matching of argument names is not allowed and e.g. `runif(1, mi = 0)` will return a message indicating that the full formal name `min` should be used. The default is set via the `gradethis.allow_partial_matching` option, or by `gradethis_setup()`.
`action`	The action to take: 1. `"pass"` provide passing `correct` feedback when the user's code matches the solution code. 2. `"fail"` provide failing `incorrect` feedback when the user's code does not match the solution code. 3. `"both"` always provide passing or failing feedback.

Value

Returns a function whose first parameter will be an environment containing objects specific to the exercise and submission (see **Available variables**). For local testing, you can create a version of the expected environment for a mock exercise submission with `mock_this_exercise()`. Calling

the returned function on the exercise-checking environment will evaluate the grade-checking `expr` and return a final grade via `graded()`.

Details

`grade_this_code()` only inspects for code differences between the student's code and the solution code. The final result of the student code and solution code is ignored. See the **Code differences** section of `code_feedback()` for implementation details on how code is determined to be different.

You can call `grade_this_code()` in two ways:

1. If you want to check the student's code without evaluating it, call `grade_this_code()` in the `*-code-check` chunk.
2. To return grading feedback in along with the resulting output of the student's code, call `grade_this_code()` in the `*-check` chunk of the exercise.

To provide the solution code, include a `*-solution` code chunk in the learnr document for the exercise to be checked. When used in this way, `grade_this_code()` will automatically find and use the student's submitted code — `.user_code` in `grade_this()` — as well as the solution code — `.solution_code` in `grade_this()`.

Custom messages

You can customize the `correct` and `incorrect` messages shown to the user by `grade_this_code()`. Both arguments accept template strings that are processed by `glue::glue()`. If you provide a custom template string, it completely overwrites the default string, but you can include the components used by the default message by adding them to your custom message.

There are four helper functions used in the default messages that you may want to include in your custom messages. To use the output of any of the following, include them inside braces in the template string. For example use `{code_feedback()}` to add the code feedback to your custom `incorrect` message.

1. `code_feedback()`: Adds feedback about the first observed difference between the student's submitted code and the model solution code. If you want to grade the student's code without providing feedback, leave `code_feedback()` out of your string.
2. `pipe_warning()`: Informs the user that their code was unpiped prior to comparison. This message is included by default to help clarify cases when the code feedback makes more sense in the unpiped context.
3. `random_praise()` and `random_encouragement()`: These praising and encouraging messages are included by default in correct and incorrect grades, by default.

See Also

`code_feedback()`, `grade_this()`, `mock_this_exercise()`

Examples

```
# For an interactive example run: gradethis_demo()

# # These are manual examples, see grading demo for `learnr` tutorial usage
```

```
grade_this_code()(
  mock_this_exercise(
    .user_code     = "sqrt(log(2))", # user submitted code
    .solution_code = "sqrt(log(1))"  # from -solution chunk
  )
)

grade_this_code()(
  mock_this_exercise(
    # user submitted code
    .user_code     = "runif(1, 0, 10)",
    # from -solution chunk
    .solution_code = "runif(n = 1, min = 0, max = 1)"
  )
)

# By default, grade_this_code() informs the user that piped code is unpiped
# when comparing to the solution
grade_this_code()(
  mock_this_exercise(
    # user submitted code
    .user_code     = "storms %>% select(year, month, hour)",
    # from -solution chunk
    .solution_code = "storms %>% select(year, month, day)"
  )
)

# By setting `correct` or `incorrect` you can change the default message
grade_this_code(
  correct = "Good work!",
  incorrect = "Not quite. {code_feedback()} {random_encouragement()}"
)(
  mock_this_exercise(
    # user submitted code
    .user_code     = "storms %>% select(year, month, hour)",
    # from -solution chunk
    .solution_code = "storms %>% select(year, month, day)"
  )
)
```

---

graded *Signal a final grade for a student's submission*

Description

`graded()` is used to signal a final grade for a submission. Most likely, you'll want to use its helper functions: `pass()`, `fail()`, `pass_if_equal()`, `fail_if_equal()`, `pass_if()` and `fail_if()`. When used in `grade_this()`, these functions signal a final grade and no further checking of the

student's submitted code is performed. See the sections below for more details about how these functions are used in grade_this().

Usage

```
graded(correct, message = NULL, ..., type = NULL, location = NULL)

pass(
  message = getOption("gradethis.pass", "Correct!"),
  ...,
  env = parent.frame(),
  praise = getOption("gradethis.pass.praise", FALSE)
)

fail(
  message = getOption("gradethis.fail", "Incorrect"),
  ...,
  env = parent.frame(),
  hint = getOption("gradethis.fail.hint", FALSE),
  encourage = getOption("gradethis.fail.encourage", FALSE)
)
```

Arguments

correct	A logical value of whether or not the checked code is correct.
message	A character string of the message to be displayed. In all grading helper functions other than graded(), message is a template string that will be processed with glue::glue().
...	Additional arguments passed to graded() or additional data to be included in the feedback object.
type, location	The type and location of the feedback object provided to **learnr**. See https://rstudio.github.io/learnr/exercises.html#Custom_checking for more details. type may be one of "auto", "success", "info", "warning", "error", or "custom". location may be one of "append", "prepend", or "replace".
env	environment to evaluate the glue message. Most users of **gradethis** will not need to use this argument.
praise	Include a random praising phrase with random_praise()? The default value of praise can be set using gradethis_setup() or the gradethis.pass.praise option.
hint	Include a code feedback hint with the failing message? This argument only applies to fail() and fail_if_equal() and the message is added using the default options of give_code_feedback() and maybe_code_feedback(). The default value of hint can be set using gradethis_setup() or the gradethis.fail.hint option.
encourage	Include a random encouraging phrase with random_encouragement()? The default value of encourage can be set using gradethis_setup() or the gradethis.fail.encourage option.

Value

`pass()` signals a *correct* submission, `fail()` signals an *incorrect* submission, and `graded()` returns a correct or incorrect submission according to the value of `correct`.

Functions

- `graded()`: Prepare and signal a graded result.
- `pass()`: Signal a *passing* grade.
- `fail()`: Signal a *failing* grade.

Usage in `grade_this()`

The `graded()` helper functions are all designed to be called from within `grade_this()`, but this has the unfortunate side-effect of making their default arguments somewhat opaque.

The helper functions follow these common patterns:

1. If you don't provide a custom `message`, the default pass or fail messages will be used. With the default **gradethis** setup, the pass message follows the pattern `{gradethis::random_praise()} Correct!` , and the fail message follows `Incorrect.{gradethis::maybe_code_feedback()} {gradethis::random_encourag`

   You can set the default message pattern using the `pass` and `fail` in `gradethis_setup()`, or the options `gradethis.pass` and `gradethis.fail`.

   In the custom `message`, you can use `glue::glue()` syntax to reference any of the available variables in `grade_this()` or that you've created in your checking code: e.g. `"Your table has {nrow(.result)} rows."`.

2. `pass_if_equal()` and `fail_if_equal()` automatically compare their first argument against the `.result` of running the student's code. `pass_if_equal()` takes this one step further and if called without any arguments will compare the `.result` to the value returned by evaluating the `.solution` code, if available.

3. All `fail` helper functions have an additional `hint` parameter. If `hint = TRUE`, a code feedback hint is added to the custom `message`. You can also control `hint` globally with `gradethis_setup()`.

4. All helper functions include an `env` parameter, that you can generally ignore. It's used internally to help `pass()` and `fail()` *et al.* find the default argument values and to build the `message` using `glue::glue()`.

Return a grade immediately

`graded()` and its helper functions are designed to short-circuit further evaluation whenever they are called. If you're familiar with writing functions in R, you can think of `graded()` (and `pass()`, `fail()`, etc.) as a special version of `return()`. If a grade is created, it is returned immediately and no more checking will be performed.

The immediate return behavior can be helpful when you have to perform complicated or long-running tests to determine if a student's code submission is correct. We recommend that you perform the easiest tests first, progressing to the most complicated tests. By taking advantage of early grade returns, you can simplify your checking code:

```
```{r}
grade_this({
  # is the answer a tibble?
  if (!inherits(.result, "tibble")) {
    fail("Your answer should be a tibble.")
  }

  # from now on we know that .result is a tibble...
  if (nrow(.result) != 5 && ncol(.result) < 2) {
    fail("Your table should have 5 rows and more than 1 column.")
  }

  # ...and now we know it has 5 rows and at least 2 columns
  if (.result[[2]][[5]] != 5) {
    fail("The value of the 5th row of the 2nd column should be 5.")
  }

  # all of the above checks have passed now.
  pass()
})
```
```

Notice that it's important to choose a final fallback grade as the last value in your grade_this() checking code. This last value is the default grade that will be given if the submission passes all other checks. If you're using the standard gradethis_setup() and you call pass() or fail() without arguments, pass() will return a random praising phrase and fail() will return code feedback (if possible) with an encouraging phrase.

See Also

Other grading helper functions: graded(), pass(), fail(), pass_if(), fail_if(), pass_if_equal(), fail_if_equal().

Examples

```
# Suppose our exercise asks the student to prepare and execute code that
# returns the value `42`. We'll use `grade_this()` to check their
# submission.
#
# Because we are demonstrating these functions inside R documentation, we'll
# save the function returned by `grade_this()` as `grader()`. Calling
# `grader()` on a mock exercise submission is equivalent to running the
# check code when the student clicks "Submit Answer" in a learnr tutorial.

grader <-
  # ```{r example-check}
  grade_this({
    # Automatically use .result to compare to an expected value
    pass_if_equal(42, "Great work!")
```

```
    # Similarly compare .result to an expected wrong value
    fail_if_equal(41, "You were so close!")
    fail_if_equal(43, "Oops, a little high there!")

    # or automatically pass if .result is equal to .solution
    pass_if_equal(message = "Great work!")

    # Be explicit if you need to round to avoid numerical accuracy issues
    pass_if_equal(x = round(.result), y = 42, "Close enough!")
    fail_if_equal(x = round(.result), y = 64, "Hmm, that's not right.")

    # For more complicated calculations, call pass() or fail()
    if (.result > 100) {
      fail("{.result} is way too high!")
    }
    if (.result * 100 == .solution) {
      pass("Right answer, but {.result} is two orders of magnitude too small.")
    }

    # Fail with a hint if student code differs from the solution
    # (Skipped automatically if there isn't a -solution chunk)
    fail_if_code_feedback()

    # Choose a default grade if none of the above have resulted in a grade
    fail()
  })
# ```

# Now lets try with a few different student submissions ----

# Correct!
grader(mock_this_exercise(.user_code = 42))

# These were close...
grader(mock_this_exercise(.user_code = 41))
grader(mock_this_exercise(.user_code = 43))

# Automatically use .solution if you have a *-solution chunk...
grader(mock_this_exercise(.user_code = 42, .solution_code = 42))

# Floating point arithmetic is tricky...
grader(mock_this_exercise(.user_code = 42.000001, .solution_code = 42))
grader(mock_this_exercise(.user_code = 64.123456, .solution_code = 42))

# Complicated checking situations...
grader(mock_this_exercise(.user_code = 101, .solution_code = 42))
grader(mock_this_exercise(.user_code = 0.42, .solution_code = 42))

# Finally fall back to the final answer...
grader(mock_this_exercise(.user_code = "20 + 13", .solution_code = "20 + 22"))
```

`gradethis_equal` *Compare the values of two objects to check whether they are equal*

Description

Compare the values of two objects to check whether they are equal

Usage

```
gradethis_equal(x = .result, y = .solution, ...)

## Default S3 method:
gradethis_equal(x, y, tolerance = sqrt(.Machine$double.eps), ...)

## S3 method for class 'list'
gradethis_equal(x, y, tolerance = sqrt(.Machine$double.eps), ...)
```

Arguments

`x`, `y`	Two objects to compare
`...`	Additional arguments passed to methods
`tolerance`	If non-`NULL`, used as threshold for ignoring small floating point difference when comparing numeric vectors. Using any non-`NULL` value will cause integer and double vectors to be compared based on their values, not their types, and will ignore the difference between `NaN` and `NA_real_`. It uses the same algorithm as `all.equal()`, i.e., first we generate `x_diff` and `y_diff` by subsetting `x` and `y` to look only locations with differences. Then we check that `mean(abs(x_diff - y_diff)) / mean(abs(y_diff))` (or just `mean(abs(x_diff - y_diff))` if `y_diff` is small) is less than `tolerance`.

Value

A logical value of length one, or an internal gradethis error.

Methods (by class)

- `gradethis_equal(default)`: The default comparison method, which uses waldo::compare
- `gradethis_equal(list)`: The comparison method for lists

Examples

```
gradethis_equal(mtcars[mtcars$cyl == 6, ], mtcars[mtcars$cyl == 6, ])
gradethis_equal(mtcars[mtcars$cyl == 6, ], mtcars[mtcars$cyl == 4, ])
```

`gradethis_error_checker` *An error checking function for use with learnr*

Description

learnr uses the checking code in `exercise.error.check.code` when the user's submission produces an error during evaluation. `gradethis_error_checker()` provides default error checking suitable for most situations where an error was *not expected*.

If a solution for the exercise is available, the user's submission will be compared to the example solution and the message to the student will include code feedback. Otherwise, the error message from R is returned.

If you *are expecting* the user to submit code that throws an error, use the `*-error-check` chunk to write custom grading code that validates that the correct error was created.

Usage

```
gradethis_error_checker(
  ...,
  hint = getOption("gradethis.fail.hint", TRUE),
  message = getOption("gradethis.error_checker.message", NULL),
  encourage = getOption("gradethis.fail.encourage", FALSE)
)
```

Arguments

`...`	Ignored but included for future compatibility.
`hint`	Include a code feedback hint with the failing message? This argument only applies to `fail()` and `fail_if_equal()` and the message is added using the default options of `give_code_feedback()` and `maybe_code_feedback()`. The default value of `hint` can be set using `gradethis_setup()` or the `gradethis.fail.hint` option.
`message`	The feedback message when an error occurred and no solution is provided for the exercise. May reference `.error` or any of the grade_this-objects. The default value is set by `gradethis_setup()`.
`encourage`	Include a random encouraging phrase with `random_encouragement()`? The default value of `encourage` can be set using `gradethis_setup()` or the `gradethis.fail.encourage` option.

Value

A checking function compatible with `gradethis_exercise_checker()`.

See Also

`gradethis_setup()`, `gradethis_exercise_checker()`

Examples

```
# The default error checker is run on an exercise that produces an error.
# In the following example, the object `b` is not defined.

# This is the error that the user's submission creates:
tryCatch(
  b,
  error = function(e) message(e$message)
)

# If you haven't provided a model solution:
gradethis_error_checker()(mock_this_exercise(b))

# If a model solution is available:
gradethis_error_checker()(mock_this_exercise(b, a))
```

---

gradethis_exercise_checker

A checker function to use with **learnr**

Description

For exercise checking, **learnr** tutorials require a function that **learnr** can use in the background to run the code in each "-check" chunk and to format the results into a format that **learnr** can display. To enable exercise checking in your learnr tutorial, attach **gradethis** with `library(gradethis)`, or call `gradethis_setup()` in the setup chunk of your tutorial. See `gradethis_demo()` to see an example learnr document that uses `gradethis_exercise_checker()`.

Usage

```
gradethis_exercise_checker(
  label = NULL,
  solution_code = NULL,
  user_code = NULL,
  check_code = NULL,
  envir_result = NULL,
  evaluate_result = NULL,
  envir_prep = NULL,
  last_value = NULL,
  stage = NULL,
  ...,
  solution_eval_fn = NULL
)
```

Arguments

`label`	Label for exercise chunk

`solution_code` Code provided within the "-solution" chunk for the exercise.

`user_code` R code submitted by the user

`check_code` Code provided within the "-check" (or "-code-check") chunk for the exercise.

`envir_result` The R environment after the execution of the chunk.

`evaluate_result`
The return value from the `evaluate::evaluate` function.

`envir_prep` A copy of the R environment before the execution of the chunk.

`last_value` The last value from evaluating the user's exercise submission.

`stage` The current stage of exercise checking.

`...` Extra arguments supplied by learnr

`solution_eval_fn`
A function taking solution `code` and an `envir` (an environment equivalent to `envir_prep`) and that will return the value of the evaluated `code`. This callback function allows grading authors to write custom solution evaluation functions for non-R exercise engines. The result of the evaluated code should be an R object that will be accessible to the grading code in .solution or .solution_all.

You may also provide a named list of solution evaluation functions to the `gradethis.exercise_checker` global option. The names of the list should match the exercise engine for which the function should be applied.

For example, for a hypothetical exercise engine `echo` that simply echoes the user's code, you could provide a `solution_eval_fn` that also just echoes the solution code:

```
options(
  gradethis.exercise_checker.solution_eval_fn = list(
    echo = function(code, envir) {
      code
    }
  )
)
```

Solution evaluation functions should determine if the solution code is missing and if so throw an error with class `error_missing_solution` (see `rlang::abort()` for help throwing this error).

Value

Returns a feedback object suitable for **learnr** tutorials with the results of the exercise grading code.

See Also

`gradethis_setup()`, `grade_this()`, `grade_this_code()`

Examples

```
## Not run:
gradethis_demo()

## End(Not run)
```

`gradethis_setup` *Setup gradethis for use within learnr*

Description

To use **gradethis** in your **learnr** tutorial, you only need to call `library(gradethis)` in your tutorial's setup chunk.

```
```{r setup}
library(learnr)
library(gradethis)
```
```

Use `gradethis_setup()` to change the default options suggested by gradethis. This function also describes in detail each of the global options available for customization in the gradethis package. Note that you most likely do not want to change the defaults values for the learnr tutorial options that are prefixed with `exercise.`. Each of the gradethis-specific arguments sets a global option with the same name, prefixed with `gradethis.`. For example, `pass` sets `gradethis.pass`.

Usage

```
gradethis_setup(
  pass = NULL,
  fail = NULL,
  ...,
  code_correct = NULL,
  code_incorrect = NULL,
  maybe_code_feedback = NULL,
  maybe_code_feedback.before = NULL,
  maybe_code_feedback.after = NULL,
  pass.praise = NULL,
  fail.hint = NULL,
  fail.encourage = NULL,
  pipe_warning = NULL,
  grading_problem.message = NULL,
  grading_problem.type = NULL,
  error_checker.message = NULL,
  allow_partial_matching = NULL,
  exercise.checker = gradethis_exercise_checker,
  exercise.timelimit = NULL,
```

```
  compare_timelimit = NULL,
  exercise.error.check.code = NULL,
  fail_code_feedback = NULL
)
```

Arguments

pass	Default message for pass(). Sets options("gradethis.pass")
fail	Default message for fail(). Sets options("gradethis.fail")
...	Arguments passed on to learnr::tutorial_options exercise.cap Caption for exercise chunk (defaults to the engine's icon or the combination of the engine and " code"). exercise.eval Whether to pre-evaluate the exercise so the reader can see some default output (defaults to FALSE). exercise.lines Lines of code for exercise editor (defaults to the number of lines in the code chunk). exercise.blanks A regular expression to be used to identify blanks in submitted code that the user should fill in. If TRUE (default), blanks are three or more underscores in a row. If FALSE, blank checking is not performed. exercise.completion Use code completion in exercise editors. exercise.diagnostics Show diagnostics in exercise editors. exercise.startover Show "Start Over" button on exercise. exercise.reveal_solution Whether to reveal the exercise solution if a solution chunk is provided.
code_correct	Default correct message for grade_this_code(). If unset, grade_this_code() falls back to the value of the gradethis.pass option. Sets the gradethis.code_correct option.
code_incorrect	Default incorrect message for grade_this_code(). If unset grade_this_code() falls back to the value of the gradethis.fail option. Sets the gradethis.code_incorrect option.
maybe_code_feedback	Logical TRUE or FALSE to determine whether maybe_code_feedback() should return code feedback, where if FALSE, maybe_code_feedback() will return an empty string. maybe_code_feedback() is used in the default messages when pass() or fail() are called without any arguments, which are set by the pass or fail arguments of gradethis_setup().
maybe_code_feedback.before, maybe_code_feedback.after	Text that should be added before or after the maybe_code_feedback() output, if any is returned. Sets the default values of the before and after arguments of maybe_code_feedback().
pass.praise	Logical TRUE or FALSE to determine whether a praising phrase should be automatically prepended to any pass() or pass_if_equal() messages. Sets the gradethis.pass.praise option.
fail.hint	Logical TRUE or FALSE to determine whether an automated code feedback hint should be shown with a fail() or fail_if_equal() message. Sets the gradethis.fail.hint option.

`fail.encourage` Logical `TRUE` or `FALSE` to determine whether an encouraging phrase should be automatically appended to any `fail()` or `fail_if_equal()` messages. Sets the `gradethis.fail.encourage` option.

`pipe_warning` The default message used in `pipe_warning()`. Sets the `gradethis.pipe_warning` option.

`grading_problem.message`
The feedback message used when a grading error occurs. Sets the `gradethis.grading_problem.messag` option.

`grading_problem.type`
The feedback type used when a grading error occurs. Must be one of `"success"`, `"info"`, `"warning"` (default), `"error"`, or `"custom"`. Sets the `gradethis.grading_problem.type` option.

`error_checker.message`
The default message used by gradethis's default error checker, `gradethis_error_checker()`. Sets the `gradethis.error_checker.message` option.

`allow_partial_matching`
Logical `TRUE` or `FALSE` to determine whether partial matching is allowed in `grade_this_code()`. Sets the `gradethis.allow_partial_matching` option.

`exercise.checker`
Function used to check exercise answers (e.g., `gradethis::grade_learnr()`).

`exercise.timelimit`
Number of seconds to limit execution time to (defaults to `30`).

`compare_timelimit`
`pass_if_equal()` and `fail_if_equal()` call `waldo::compare()` internally. This helps ensure an accurate comparison, but sometimes takes a long time. `compare_timelimit` is the time limit in seconds for the execution of `waldo::compare()` (defaults to 80% of `exercise.timelimit`). If the time limit is exceeded, `identical()` is used instead of `waldo::compare()`.

`exercise.error.check.code`
A string containing R code to use for checking code when an exercise evaluation error occurs (e.g., `"gradethis::grade_code()"`).

`fail_code_feedback`
Deprecated. Use `maybe_code_feedback`.

Value

Invisibly returns the global options as they were prior to setting them with `gradethis_setup()`.

Global package options

These global package options can be set by `gradethis_setup()` or by directly setting the global option. The default values set for each option when gradethis is loaded are shown below.

Option	Default Value
`gradethis.pass`	`"{gradethis::random_praise()} Correct!"`
`gradethis.pass.praise`	`FALSE`
`gradethis.fail`	`"Incorrect.{gradethis::maybe_code_feedback()} {gradethis::rando`

```
gradethis.fail.hint                   FALSE
gradethis.fail.encourage              FALSE
gradethis.maybe_code_feedback         TRUE
gradethis.maybe_code_feedback.before  " "
gradethis.maybe_code_feedback.after   NULL
gradethis.code_correct                NULL
gradethis.code_incorrect              "{gradethis::pipe_warning()}{gradethis::code_feedback()} {grade
gradethis.pipe_warning                "I see that you are using pipe operators (e.g. %>%), so I want to let
gradethis.grading_problem.message     "A problem occurred with the grading code for this exercise."
gradethis.grading_problem.type        "warning"
gradethis.allow_partial_matching      NULL
gradethis.error_checker.message       "An error occurred with your code:\n\n“‘\n{.error$message}\n“‘\n\
gradethis.compare_timelimit           NULL
```

See Also

gradethis_exercise_checker()

Examples

```
# Not run in package documentation because this function changes global opts
if (FALSE) {
  old_opts <- gradethis_setup(
    pass = "Great work!",
    fail = "{random_encouragement()}"
  )
}

# Use getOption() to see the default value
getOption("gradethis.pass")
getOption("gradethis.maybe_code_feedback")
```

---

mock_this_exercise *Mock a user submission to an exercise*

Description

This function helps you test your grade_this() and grade_this_code() logic by helping you quickly create the environment that these functions expect when used to grade a user submission to an exercise in a **learnr** tutorial.

Usage

```
mock_this_exercise(
  .user_code,
  .solution_code = NULL,
  ...,
```

```
  .label = "mock",
  .engine = "r",
  .stage = "check",
  .result = rlang::missing_arg(),
  setup_global = NULL,
  setup_exercise = NULL
)
```

Arguments

`.user_code`	A single string or expression in braces representing the user submission to this exercise.
`.solution_code`	An optional single string or expression in braces representing the solution code to this exercise.
`...`	Ignored
`.label`	The label of the mock exercise, defaults to `"mock"`.
`.engine`	The engine of the mock exercise. If the engine is not `"r"`, then `.result` must be provided explicitly since `mock_this_exercise()` cannot evaluate the `.user_code`.
`.stage`	The stage of the exercise evaluation, defaults to `"check"`. **learnr** stages are `"code_check"`, `"check"` or `"error_check"`. When gradethis is used outside of learnr, this variable is typically `NULL`.
`.result`	The result of the evaluation of the `.user_code`. If the `.engine` is `"r"`, the result will be prepared automatically by evaluating the user code.
`setup_global`	An optional single string or expression in braces representing the global `setup` chunk code.
`setup_exercise`	An optional single string or expression in braces representing the code in the exercise's setup chunk(s).

Value

Returns the checking environment that is expected by `grade_this()` and `grade_this_code()`. Both of these functions themselves return a function that gets called on the checking environment. In other words, the object returned by this function can be passed to the function returned from either `grade_this()` or `grade_this_code()` to test the grading logic used in either.

Examples

```
# First we'll create a grading function with grade_this(). The user's code
# should return the value 42, and we have some specific messages if they're
# close but miss this target. Otherwise, we'll fall back to the default fail
# message, which will include code feedback.
this_grader <-
  grade_this({
    pass_if_equal(42, "Great Work!")
    fail_if_equal(41, "You were so close!")
    fail_if_equal(43, "Oops, just missed!")
    fail()
  })
```

```
# Our first mock submission is almost right...
this_grader(mock_this_exercise(.user_code = 41, .solution_code = 42))

# Our second mock submission is a little too high...
this_grader(mock_this_exercise(.user_code = 43, .solution_code = 42))

# A third submission takes an unusual path, but arrives at the right answer.
# Notice that you can use braces around an expression.
this_grader(
  mock_this_exercise(
    .user_code = {
      x <- 31
      y <- 11
      x + y
    },
    .solution_code = 42
  )
)

# Our final submission changes the prompt slightly. Suppose we have provided
# an `x` object in our global setup with a value of 31. We also have a `y`
# object that we create for the user in the exercise setup chunk. We then ask
# the student to add `x` and `y`. What happens if the student subtracts
# instead? That's what this mock submission tests:
this_grader(
  mock_this_exercise(
    .user_code = x - y,
    .solution_code = x + y,
    setup_global = x <- 31,
    setup_exercise = y <- 11
  )
)
```

`pass_if` *Signal a passing or failing grade if a condition is TRUE*

Description

`pass_if()` and `fail_if()` both create passing or failing grades if a given condition is `TRUE`. See `graded()` for more information on **gradethis** grade-signaling functions.

These functions are also used in legacy **gradethis** code, in particular in the superseded function `grade_result()`. While previous versions of **gradethis** allowed the condition to be determined by a function or formula, when used in `grade_this()` the condition must be a logical `TRUE` or `FALSE`.

Usage

```
pass_if(
  cond,
```

```
  message = NULL,
  ...,
  env = parent.frame(),
  praise = getOption("gradethis.pass.praise", FALSE),
  x = deprecated()
)

fail_if(
  cond,
  message = NULL,
  ...,
  env = parent.frame(),
  hint = getOption("gradethis.fail.hint", FALSE),
  encourage = getOption("gradethis.fail.encourage", FALSE),
  x = deprecated()
)
```

Arguments

cond	A logical value or an expression that will evaluate to a TRUE or FALSE value. If the value is TRUE, or would be considered TRUE in an `if (cond)` statement, then a passing or failing grade is returned to the user.
message	A character string of the message to be displayed. In all grading helper functions other than graded(), message is a template string that will be processed with glue::glue().
...	Passed to graded() in grade_this().
env	environment to evaluate the glue message. Most users of **gradethis** will not need to use this argument.
praise	Include a random praising phrase with random_praise()? The default value of praise can be set using gradethis_setup() or the gradethis.pass.praise option.
x	Deprecated. Replaced with cond.
hint	Include a code feedback hint with the failing message? This argument only applies to fail() and fail_if_equal() and the message is added using the default options of give_code_feedback() and maybe_code_feedback(). The default value of hint can be set using gradethis_setup() or the gradethis.fail.hint option.
encourage	Include a random encouraging phrase with random_encouragement()? The default value of encourage can be set using gradethis_setup() or the gradethis.fail.encourage option.

Value

pass_if() and fail_if() signal a correct or incorrect grade if the provided condition is TRUE.

Functions

- `pass_if()`: Pass if cond is `TRUE`.
- `fail_if()`: Fail if cond is `TRUE`.

See Also

Other grading helper functions: `graded()`, `pass()`, `fail()`, `pass_if()`, `fail_if()`, `pass_if_equal()`, `fail_if_equal()`.

Examples

```
# Suppose the prompt is to find landmasses in `islands` with land area of
# less than 20,000 square miles. (`islands` reports land mass in units of
# 10,000 sq. miles.)

grader <-
  # ```{r example-check}
  grade_this({
    fail_if(any(is.na(.result)), "You shouldn't have missing values.")

    diff_len <- length(.result) - length(.solution)
    fail_if(diff_len < 0, "You missed {abs(diff_len)} island(s).")
    fail_if(diff_len > 0, "You included {diff_len} too many islands.")

    pass_if(all(.result < 20), "Great work!")

    # Fall back grade
    fail()
  })
# ```

.solution <-
  # ```{r example-solution}
  islands[islands < 20]
# ```

# Peek at the right answer
.solution

# Has missing values somehow
grader(mock_this_exercise(islands["foo"], !!.solution))

# Has too many islands
grader(mock_this_exercise(islands[islands < 29], !!.solution))

# Has too few islands
grader(mock_this_exercise(islands[islands < 16], !!.solution))

# Just right!
grader(mock_this_exercise(islands[islands < 20], !!.solution))
```

pass_if_equal *Signal a passing or failing grade if two values are equal*

Description

`pass_if_equal()`, `fail_if_equal()`, and `fail_if_not_equal()` are three `graded()` helper functions that signal a passing or a failing grade based on the whether two values are equal. They are designed to easily compare the returned value of the student's submitted code with the value returned by the solution or another known value:

- Each function finds and uses `.result` as the default for `x`, the first item in the comparison. `.result` is the last value returned from the user's submitted code.
- `pass_if_equal()` additionally finds and uses `.solution` as the default expected value `y`.

See `graded()` for more information on **gradethis** grade-signaling functions.

Usage

```
pass_if_equal(
  y = .solution,
  message = getOption("gradethis.pass", "Correct!"),
  x = .result,
  ...,
  env = parent.frame(),
  tolerance = sqrt(.Machine$double.eps),
  praise = getOption("gradethis.pass.praise", FALSE)
)

fail_if_equal(
  y,
  message = getOption("gradethis.fail", "Incorrect"),
  x = .result,
  ...,
  env = parent.frame(),
  tolerance = sqrt(.Machine$double.eps),
  hint = getOption("gradethis.fail.hint", FALSE),
  encourage = getOption("gradethis.fail.encourage", FALSE)
)

fail_if_not_equal(
  y,
  message = getOption("gradethis.fail", "Incorrect"),
  x = .result,
  ...,
  env = parent.frame(),
  tolerance = sqrt(.Machine$double.eps),
  hint = getOption("gradethis.fail.hint", FALSE),
```

```
  encourage = getOption("gradethis.fail.encourage", FALSE)
)
```

Arguments

y	The expected value against which x is compared using gradethis_equal(x, y). In pass_if_equal(), if no value is provided, the exercise .solution (i.e. the result of evaluating the code in the exercise's *-solution chunk) will be used for the comparison. If the exercise uses multiple solutions with *different results*, set y = .solution_all. In this case, pass_if_equal() will test each of the solutions and provide a passing grade if x matches *any* values contained in y. Note that if the exercise has multiple solutions but they all return the same result, it will be faster to use the default value of y = .solution.
message	A character string of the message to be displayed. In all grading helper functions other than graded(), message is a template string that will be processed with glue::glue().
x	First item in the comparison. By default, when used inside grade_this(), x is automatically assigned the value of .result — in other words the result of running the student's submitted code. x is not the first argument since you will often want to compare the final value of the student's submission against a specific value, y.
...	Additional arguments passed to graded()
env	environment to evaluate the glue message. Most users of **gradethis** will not need to use this argument.
tolerance	If non-NULL, used as threshold for ignoring small floating point difference when comparing numeric vectors. Using any non-NULL value will cause integer and double vectors to be compared based on their values, not their types, and will ignore the difference between NaN and NA_real_. It uses the same algorithm as all.equal(), i.e., first we generate x_diff and y_diff by subsetting x and y to look only locations with differences. Then we check that mean(abs(x_diff - y_diff)) / mean(abs(y_diff)) (or just mean(abs(x_diff - y_diff)) if y_diff is small) is less than tolerance.
praise	Include a random praising phrase with random_praise()? The default value of praise can be set using gradethis_setup() or the gradethis.pass.praise option.
hint	Include a code feedback hint with the failing message? This argument only applies to fail() and fail_if_equal() and the message is added using the default options of give_code_feedback() and maybe_code_feedback(). The default value of hint can be set using gradethis_setup() or the gradethis.fail.hint option.
encourage	Include a random encouraging phrase with random_encouragement()? The default value of encourage can be set using gradethis_setup() or the gradethis.fail.encourage option.

Value

Returns a passing or failing grade if `x` and `y` are equal.

Functions

- `pass_if_equal()`: Signal a *passing* grade only if `x` and `y` are equal.
- `fail_if_equal()`: Signal a *failing* grade only if `x` and `y` are equal.
- `fail_if_not_equal()`: Signal a *failing* grade if `x` and `y` are *not* equal.

Comparing with Multiple Solutions

If your exercise includes multiple solutions that are variations of the same task — meaning that all solutions achieve the same result — you can call `pass_if_equal()` without changing any defaults to compare the result of the student's submission to the common solution result. After checking if any solution matches, you can perform additional checks or you can call `fail()` with the default message or with `hint = TRUE`. `fail()` will automatically provide code feedback for the most likely solution.

By default, `pass_if_equal()` will compare .result with .solution, or the final value returned by the entire `-solution` chunk (in other words, the last solution). This default behavior covers both exercises with a single solution and exercises with multiple solutions that all return the same value.

When your exercise has multiple solutions with **different results**, `pass_if_equal()` can compare the student's .result to each of the solutions in .solution_all, returning a passing grade when the result matches any of the values returned by the set of solutions. You can opt into this behavior by calling

```
pass_if_equal(.solution_all)
```

Note that this causes `pass_if_equal()` to evaluate each of the solutions in the set, and may increase the computation time.

Here's a small example. Suppose an exercise asks students to filter `mtcars` to include only cars with the same number of cylinders. Students are free to pick cars with 4, 6, or 8 cylinders, and so your `-solution` chunk would include this code (ignoring the `ex_solution` variable, the chunk would contain the code in the string below):

```
ex_solution <- "
# four cylinders ----
mtcars[mtcars$cyl == 4, ]

# six cylinders ----
mtcars[mtcars$cyl == 6, ]

# eight cylinders ----
mtcars[mtcars$cyl == 8, ]
"
```

In the `-check` chunk, you'd call `grade_this()` and ask `pass_if_equal()` to compare the student's .result to .solution_all (all the solutions).

```
ex_check <- grade_this({
  pass_if_equal(
    y = .solution_all,
    message = "The cars in your result all have {.solution_label}!"
   )

  fail()
})
```

What happens when a student submits one of these solutions? This function below mocks the process of a student submitting an attempt.

```
student_submits <- function(code) {
  withr::local_seed(42)
  submission <- mock_this_exercise(!!code, !!ex_solution)
  ex_check(submission)
}
```

If they submit code that returns one of the three possible solutions, they receive positive feedback.

```
student_submits("mtcars[mtcars$cyl == 4, ]")
#> <gradethis_graded: [Correct]
#>   The cars in your result all have four cylinders!
#> >
student_submits("mtcars[mtcars$cyl == 6, ]")
#> <gradethis_graded: [Correct]
#>   The cars in your result all have six cylinders!
#> >
```

Notice that the solution label appears in the feedback message. When `pass_if_equal()` picks a solution as correct, three variables are made available for use in the glue string provided to `message`:

- `.solution_label`: The heading label of the matching solution
- `.solution_code`: The code of the matching solution
- `.solution`: The value of the evaluated matching solution code

If the student submits incorrect code, `pass_if_equal()` defers to later grading code.

```
student_submits("mtcars[mtcars$cyl < 8, ]")
#> <gradethis_graded: [Incorrect]
#>   Incorrect. In `mtcars[mtcars$cyl < 8, ]`, I expected you to call `==`
#>   where you called `<`. Please try again.
#> >
```

Here, because `fail()` provides `code_feedback()` by default, and because `code_feedback()` is also aware of the multiple solutions for this exercise, the code feedback picks the *eight cylinders* solution and gives advice based on that particular solution.

See Also

Other grading helper functions: `graded()`, `pass()`, `fail()`, `pass_if()`, `fail_if()`, `pass_if_equal()`, `fail_if_equal()`.

Examples

```
# Suppose our prompt is to find the cars in `mtcars` with 6 cylinders...

grader <-
  # ```{r example-check}
  grade_this({
    # Automatically pass if .result equal to .solution
    pass_if_equal()

    fail_if_equal(mtcars[mtcars$cyl == 4, ], message = "Not four cylinders")
    fail_if_equal(mtcars[mtcars$cyl == 8, ], message = "Not eight cylinders")

    # Default to failing grade with feedback
    fail()
  })
# ```

.solution <-
  # ```{r example-solution}
  mtcars[mtcars$cyl == 6, ]
# ```

# Correct!
grader(mock_this_exercise(mtcars[mtcars$cyl == 6, ], !!.solution))

# These fail with specific messages
grader(mock_this_exercise(mtcars[mtcars$cyl == 4, ], !!.solution))
grader(mock_this_exercise(mtcars[mtcars$cyl == 8, ], !!.solution))

# This fails with default feedback message
grader(mock_this_exercise(mtcars[mtcars$mpg == 8, ], !!.solution))
```

---

pipe_warning *Inform the user about how gradethis interprets piped code*

Description

Creates a warning message when user code contains the `%>%`. When feedback is automatically generated via `code_feedback()` or in `grade_this_code()`, this message attempts to contextualize feedback that might make more sense when referenced against an un-piped version of the student's code.

Usage

```
pipe_warning(message = getOption("gradethis.pipe_warning"), .user_code = NULL)
```

Arguments

message	A glue string containing the message. The default value is set with the gradethis.pipe_warning option.
.user_code	The user's submitted code, found in env if NULL

Value

Returns a string containing the pipe warning message, or an empty string if the .user_code does not contain a pipe, if the .user_code is also empty, or if the message is NULL.

Options

- gradethis.pipe_warning: The default pipe warning message is set via this option.

Glue Variables

The following variables may be used in the glue-able message:

- .user_code: The student's original submitted code.
- .user_code_unpiped: The unpiped version of the student's submitted code.

Examples

```
# The default `pipe_warning()` message:
getOption("gradethis.pipe_warning")

# Let's consider two versions of the user code
user_code <-  "penguins %>% pull(year) %>% min(year)"
user_code_unpiped <- "min(pull(penguins, year), year)"

# A `pipe_warning()` is created when the user's code contains `%>%`
pipe_warning(.user_code = user_code)

# And no message is created when the user's code in un-piped
pipe_warning(.user_code = user_code_unpiped)

# Typically, this warning is only introduced when giving code feedback
# for an incorrect submission. Here we didn't expect `year` in `min()`.
submission <- mock_this_exercise(
  .user_code = !!user_code,
  .solution_code = "penguins %>% pull(year) %>% min()"
)

grade_this_code()(submission)
```

praise *Random praise and encouragement*

Description

Generate a random praise or encouragement phrase. These functions are designed for use within `pass()` or `fail()` messages, or anywhere else that **gradethis** provides feedback to the student.

Usage

```
random_praise()

random_encouragement()

give_praise(expr, ..., location = "before", before = NULL, after = NULL)

give_encouragement(expr, ..., location = "after", before = NULL, after = NULL)
```

Arguments

`expr`	A `graded()` grade or helper function, or a grading function — like `grade_this()` or `grade_result()` — or a character string. Praise will be added to any passing grades and encouragement will be added to any failing grade. If `expr` is a character string, then `"{random_praise()}"` or `"{random_encouragement()}"` is pasted before or after the string according to `location`.
`...`	Ignored.
`location`	Should the praise or encouragement be added before or after the grade message?
`before, after`	Text to be added before or after the praise or encouragement phrase.

Value

- `random_praise()` and `random_encouragement()` each return a length-one string with a praising or encouraging phrase.
- `give_praise()` and `give_encouragement()` add praise or encouragement phrases to passing and failing grades, respectively.

Functions

- `random_praise()`: Random praising phrase
- `random_encouragement()`: Random encouraging phrase
- `give_praise()`: Add praising message to a passing grade.
- `give_encouragement()`: Add encouraging message to a failing grade.

Examples

```
replicate(5, glue::glue("Random praise: {random_praise()}"))
replicate(5, glue::glue("Random encouragement: {random_encouragement()}"))

# give_praise() adds praise to passing grade messages
give_praise(pass("That's absolutely correct."))

# give_encouragement() encouragement to failing grade messages
give_encouragement(fail("Sorry, but no."))
```

`user_object` *Functions for interacting with objects created by student and solution code*

Description

Functions for interacting with objects created by student and solution code

Usage

```
user_object_get(x, mode = "any", ..., check_env = parent.frame())

solution_object_get(x, mode = "any", ..., check_env = parent.frame())

user_object_exists(x, mode = "any", ..., check_env = parent.frame())

solution_object_exists(x, mode = "any", ..., check_env = parent.frame())

user_object_list(
  mode = "any",
  exclude_envir = .envir_prep,
  ...,
  check_env = parent.frame()
)

solution_object_list(
  mode = "any",
  exclude_envir = .envir_prep,
  ...,
  check_env = parent.frame()
)
```

Arguments

x	An object name, given as a quoted character string.
mode	character specifying the mode of objects to consider. Passed to exists and get.

exclude_envir	An environment. Objects that appear in exclude_envir will be excluded from results. Defaults to .envir_prep. Use exclude_envir = NULL to include all objects.
...	Additional arguments passed to underlying functions: • For user_object_exists() and solution_object_exists(), exists() • For user_object_get() and solution_object_get(), get() • For user_object_list() and solution_object_list(), ls.str()
check_env	The environment from which to retrieve .envir_result and .envir_prep. Most users of **gradethis** will not need to use this argument.

Value

For user_object_get() and solution_object_get(), the object. If the object is not found, an error.

For user_object_exists() and solution_object_exists(), a TRUE/FALSE value.

For user_object_list() and solution_object_list(), a character vector giving the names of the objects created by the student or solution code.

Examples

```
user_code <- quote({
  # ```{r example}
  x <- "I'm student code!"
  y <- list(1, 2, 3)
  z <- function() print("Hello World!")
  # ```
})

solution_code <- quote({
  # ```{r example-solution}
  x <- "I'm solution code!"
  y <- list("a", "b", "c")
  z <- function() print("Goodnight Moon!")
  # ```
})

exercise <- mock_this_exercise(!!user_code, !!solution_code)

with_exercise(exercise, user_object_list())
with_exercise(exercise, user_object_exists("x"))
with_exercise(exercise, user_object_get("x"))

with_exercise(exercise, solution_object_list())
with_exercise(exercise, solution_object_exists("x"))
with_exercise(exercise, solution_object_get("x"))

# Use `mode` to find only objects of a certain type ----

with_exercise(exercise, user_object_list(mode = "character"))
with_exercise(exercise, user_object_list(mode = "list"))
```

```
with_exercise(exercise, user_object_list(mode = "function"))

with_exercise(exercise, user_object_exists("x", mode = "character"))
with_exercise(exercise, user_object_exists("y", mode = "character"))

with_exercise(exercise, user_object_get("z", mode = "function"))

# By default, `user_object_list()` ignores objects created by setup chunks ----

setup_code <- rlang::expr({
  # ```{r example-setup}
  setup_data <- mtcars
  # ```
})

setup_exercise <- mock_this_exercise(
  !!user_code, !!solution_code, setup_exercise = !!setup_code
)

with_exercise(setup_exercise, user_object_list())

## You can disable this by setting `exclude_envir = NULL` ----

with_exercise(setup_exercise, user_object_list(exclude_envir = NULL))
```

`with_exercise` *Run an expression as if it were in an exercise's* `grade_this()` *block*

Description

This function is not intended to be used within grading code, but may be helpful for testing grading code.

Usage

```
with_exercise(exercise, expr)
```

Arguments

`exercise`	An exercise, as created by `mock_this_exercise()`
`expr`	An unquoted expression

Value

The value of `grade_this(<expr>)(exercise)`

Examples

```
exercise <- mock_this_exercise(.user_code = "2", .solution_code = "1 + 1")

with_exercise(exercise, pass_if_equal())
with_exercise(exercise, fail_if_code_feedback())
```

Index