

Package: garnish (via r-universe)

November 1, 2026

Title Python-Style Decorators for R

Version 0.1.0

Description Popular Python decorators ported to R with the 'adorn' framework: retrying, caching, context managers and argument validation.

License MIT + file LICENSE

URL <https://dyfanjones.github.io/garnish/>,
<https://github.com/DyfanJones/garnish>

BugReports <https://github.com/DyfanJones/garnish/issues>

Depends R (>= 4.4.0)

Imports adorn, S7 (>= 0.2.2.9000), rlang, cachem, stats, withr

Remotes DyfanJones/adorn, RConsortium/S7

Suggests litedown, qs2, testthat (>= 3.0.0)

Config/testthat/edition 3

Config/roxygen2/version 8.1.1

Encoding UTF-8

Roxygen list(markdown = TRUE)

Collate 'cache.R' 'circuit_breaker.R' 'contextmanager.R'
'rate_limit.R' 'reexport-S7.R' 'reexport-adorn.R'
'retry_utils.R' 'retry_after.R' 'retry_before.R'
'retry_retry.R' 'retry_wait.R' 'retry_stop.R' 'retry.R'
'retry_extend.R' 'validate.R' 'zzz.R'

VignetteBuilder litedown

Repository <https://dyfanjones.r-universe.dev>

Date/Publication 2026-10-31 23:15:35 UTC

RemoteUrl <https://github.com/DyfanJones/garnish>

RemoteRef HEAD

RemoteSha 37c900aa0f4935e1804caa9dacffa21adc1a456c

Contents

after_sleep_base	2
after_sleep_none	4
as_decorator	5
before_sleep_none	5
circuit_breaker	6
contextmanager	7
lru_cache	8
rate_limit	10
retry	11
retry_all	12
stop_never	13
validate_args	14
wait_fixed	15
with_context	16
Index	18

after_sleep_base	<i>Write your own stop, wait, retry and hook strategies</i>
------------------	---

Description

`retry()` is built from five kinds of strategy. Each has an abstract base class and a generic function, both exported, so you can add your own the same way the built-in ones are written with S7:

Usage

```

after_sleep_base()

after_sleep_fn(class, state, ...)

before_sleep_base()

before_sleep_fn(class, state, ...)

retry_base()

retry_fn(class, outcome, ...)

wait_base()

wait_fn(class, attempt, prev_wait = NULL, ...)

stop_base()

stop_fn(class, state, ...)
```

Arguments

<code>class</code>	A strategy object. Methods are dispatched on this argument.
<code>state, outcome, attempt, prev_wait</code>	Described above.
<code>...</code>	Passed to methods.

Details

retry() argument	Base class	Generic	The method returns
<code>stop</code>	<code>stop_base</code>	<code>stop_fn(class, state)</code>	TRUE to give up
<code>wait</code>	<code>wait_base</code>	<code>wait_fn(class, attempt, prev_wait)</code>	seconds to sleep
<code>retry</code>	<code>retry_base</code>	<code>retry_fn(class, outcome)</code>	TRUE to try again
<code>before_sleep</code>	<code>before_sleep_base</code>	<code>before_sleep_fn(class, state)</code>	nothing
<code>after_sleep</code>	<code>after_sleep_base</code>	<code>after_sleep_fn(class, state)</code>	nothing

To add a strategy, make a class that inherits from the base class, then register a method for the generic with `S7::method<-(.)`. Create the object with `new_object()` on an instance of the base class, as the built-in strategies do. Load S7 with `library(S7)` to get `new_class()`, `new_object()` and `method<-`.

Strategies of the same kind compose with the built-in ones. A new wait works in `wait_chain()` and with `+`, and a new stop works in `stop_any()` and `stop_all()`.

If you register methods from another package, call `S7::methods_register()` in its `.onLoad()`.

Value

`stop_fn()` and `retry_fn()` return a single TRUE or FALSE, `wait_fn()` returns a number of seconds, and the hook generics return nothing useful.

Arguments passed to methods

- `state`, for `stop_fn()`, `before_sleep_fn()` and `after_sleep_fn()`, is a list describing the attempt that just failed: `attempt` (1-based), `wait` (seconds `retry()` is about to sleep), `elapsed` (seconds since the first attempt), `error` (the condition caught, or NULL if `retry_if_result()` rejected a returned value) and `value` (the value returned, or NULL if the call errored).
- `attempt` and `prev_wait`, for `wait_fn()`, are the 1-based attempt number and the previous sleep in seconds (NULL on the first retry).
- `outcome`, for `retry_fn()`, is `list(value, error)` for the attempt that just finished. `error` is NULL when the call returned normally, and `value` is NULL when it errored.

Examples

```
library(S7)

# Wait `step` seconds longer after each failure
wait_linear <- new_class(
```

```

    "wait_linear",
    parent = wait_base,
    properties = list(step = new_property(class_numeric, default = 1))
  )

  method(wait_fn, wait_linear) <- function(class, attempt, prev_wait = NULL, ...) {
    class@step * attempt
  }

  f <- retry(
    stop = stop_after_attempt(3),
    wait = wait_linear(0.01) + wait_fixed(0.01)
  ) %% function() stop("flaky")
  try(f(), silent = TRUE)

```

after_sleep_none	<i>Run a hook after <code>retry()</code> sleeps</i>
------------------	---

Description

- `after_sleep_none()` does nothing.
- `after_sleep_log()` passes a message describing the attempt to logger.

Usage

```
after_sleep_none()
```

```
after_sleep_log(logger = message)
```

Arguments

`logger` Function taking a single message string. Defaults to `message()`.

Value

A hook to pass to `retry()`.

Examples

```
after_sleep_log(function(msg) cat(msg, "\n"))
```

as_decorator	<i>Use a context manager as a decorator</i>
--------------	---

Description

Decorator form of a `contextmanager()`: the decorated function runs inside the context, and cleanup runs when it returns or errors. As in Python's `ContextDecorator`, the context's value is not passed to the function.

Usage

```
as_decorator(ctx, ...)
```

Arguments

ctx	A context manager.
...	Arguments for ctx.

Value

A decorator.

Examples

```
local_tempdir := contextmanager %% function() {
  path <- tempfile()
  dir.create(path)
  teardown(unlink(path, recursive = TRUE))
  path
}
g := as_decorator(local_tempdir) %% function() "ran inside context"
g()
```

before_sleep_none	<i>Run a hook before <code>retry()</code> sleeps</i>
-------------------	--

Description

- `before_sleep_none()` does nothing.
- `before_sleep_log()` passes a message describing the attempt to logger.

Usage

```
before_sleep_none()
```

```
before_sleep_log(logger = message)
```

Arguments

logger Function taking a single message string. Defaults to `message()`.

Value

A hook to pass to `retry()`.

Examples

```
before_sleep_log(function(msg) cat(msg, "\n"))
```

circuit_breaker	<i>Stop calling a function that keeps failing</i>
-----------------	---

Description

Decorator that stops calling a function after it fails repeatedly, so callers fail fast instead of waiting on something that is down. It has three states:

Usage

```
circuit_breaker(f, failures = 5, reset_after = 30)
```

Arguments

f	The function to decorate. Usually supplied by <code>%%</code> rather than directly.
failures	Number of consecutive errors that open the circuit. A whole number of at least 1. Defaults to 5.
reset_after	Seconds the circuit stays open before a trial call. A number greater than 0. Defaults to 30.

Details

- **closed**: calls go through. Consecutive errors are counted.
- **open**: after failures consecutive errors, calls fail immediately with a `garnish_circuit_open` error, without running the function, for `reset_after` seconds.
- **half-open**: once the cooldown has passed, the next call runs as a trial. If it succeeds the circuit closes. If it fails the circuit opens again for another cooldown.

Only errors count as failures. A call that returns normally, whatever its value, resets the count. Each decorated function keeps its own state.

Stack it outside `retry()` so an open circuit rejects a call before any retries happen:

```
fetch := circuit_breaker(failures = 3, reset_after = 30) %%
  retry(stop = stop_after_attempt(3)) %%
  function(url) { ... }
```

Value

A decorator.

Examples

```
n <- 0
flaky <- function() {
  n <<- n + 1
  stop("service down")
}
f := circuit_breaker(failures = 2, reset_after = 60) %% flaky
try(f())
try(f())
try(f()) # rejected without calling flaky()
n # still 2
```

contextmanager

Write setup and teardown once

Description

Port of Python's `contextlib.contextmanager`. R has no `yield`, so instead of suspending the function the body calls `teardown()` right after acquiring each resource and returns the resource. From that one definition you get:

Usage

```
contextmanager(f)
```

```
teardown(expr)
```

Arguments

<code>expr</code>	Code that releases the resource. Evaluated in the setup function's environment when the context ends, so it can use the setup's local variables.
<code>f</code>	The function to decorate. Usually supplied by <code>%%</code> rather than directly.

Details

- a `local_` form: the function itself. Cleanup runs when the *calling* function exits, like `withr::local_*`.
- a `with_` form: `with_context()` runs a block, then cleans up.
- a decorator: `as_decorator()` runs a function inside the context.

The decorated function takes `...`, passed on to the setup function, and a `.local_envir` argument naming the frame whose exit triggers cleanup. It defaults to the caller, or, when called from another context manager's setup, to that context's `.local_envir`, so composed contexts last as long as the outer one.

teardown(expr) registers expr with `withr::defer()` on that frame. Cleanups run last in, first out, and the ones already registered still run if the rest of the setup fails. `teardown()` finds the frame by searching the call stack for the nearest running context manager and reading its `.local_envir`.

Value

contextmanager is a decorator. `teardown()` is called for its side effect.

See Also

`with_context()`, `as_decorator()`

Examples

```
local_tempdir := contextmanager %% function(pattern = "tmp") {
  path <- tempfile(pattern)
  dir.create(path)
  teardown(unlink(path, recursive = TRUE))
  path
}

f <- function() {
  d <- local_tempdir()
  writeLines("hi", file.path(d, "x.txt"))
  d
}
d <- f()
dir.exists(d) # cleaned up when f() exited
```

lru_cache	Cache function results
-----------	------------------------

Description

Decorators that remember a function’s results, keyed on the matched arguments (including defaults).

Usage

```
lru_cache(f, maxsize = 1024 * 1024^2)

cache(f)

disk_cache(f, dir, maxsize = 1024 * 1024^2, format = c("rds", "qs2"))
```

Arguments

<code>f</code>	The function to decorate. Usually supplied by <code>%%</code> rather than directly.
<code>maxsize</code>	Maximum cache size in bytes. Defaults to 1 GiB.
<code>format</code>	File format for <code>disk_cache()</code> : "rds" (the default) or "qs2", which needs the qs2 package.
<code>dir</code>	Directory for <code>disk_cache()</code> to keep its files in. It is created if needed. There is no default, so persistence is always a choice.

Details

- `lru_cache()` evicts the least recently used entries when the cache is full. Port of Python's `functools.lru_cache`.
- `cache()` never evicts. Port of Python's `functools.cache`.
- `disk_cache()` stores results as files in `dir`, so they survive the R session and can be shared between sessions. Like `joblib.Memory.cache`.

Entries are keyed on the function's formals and body as well as its arguments, so editing the function invalidates them. Changes to anything it calls do not. `disk_cache()` saves results with [saveRDS\(\)](#), or with the qs2 package when `format = "qs2"`, which is usually faster and smaller for large objects. Neither round-trips environments or external pointers. Files from the two formats are not shared, so changing format for an existing `dir` starts with an empty cache.

Call `lru_cache(maxsize)` to build a decorator, then apply it with `%%` or by calling it on a function. `cache` takes no parameters, so it is applied directly.

Value

A decorator.

Examples

```
slow_square <- lru_cache() %% function(x) {
  Sys.sleep(0.1)
  x^2
}
slow_square(4) # slow
slow_square(4) # cached

slow_cube <- cache %% function(x) {
  Sys.sleep(0.1)
  x^3
}
slow_cube(2) # slow
slow_cube(2) # cached

dir <- tempfile("cache")
slow_cube <- disk_cache(dir) %% function(x) {
  Sys.sleep(0.1)
  x^3
}
```

```
slow_cube(2) # slow, and saved to `dir`
slow_cube(2) # cached
```

rate_limit

Limit how often a function can be called

Description

Decorator that allows at most `calls` calls in any window of `period` seconds. A call that would exceed the limit sleeps until the oldest call in the window has aged out. Calls are counted when they start, so a call that errors still uses up its slot, as it does for an API.

Usage

```
rate_limit(f, calls, period = 1)
```

Arguments

<code>f</code>	The function to decorate. Usually supplied by <code>%%</code> rather than directly.
<code>calls</code>	Maximum number of calls in one window. A whole number of at least 1.
<code>period</code>	Length of the window in seconds. Defaults to 1.

Details

Call `rate_limit(calls, period)` to build a decorator, then apply it with `%%` or by calling it on a function. Each decorated function keeps its own record of calls, so two functions decorated separately have separate limits.

Stack it inside `retry()` so that every attempt is throttled, including retries. Stacking it outside would only throttle whole calls, and retries could still exceed the limit.

Value

A decorator.

Examples

```
f <- rate_limit(calls = 3, period = 1) %% function(x) x
vapply(1:3, f, numeric(1)) # immediate
system.time(f(4)) # waits for the window to clear

# Throttle every attempt, retries included
fetch <- retry(stop = stop_after_attempt(3)) %%
  rate_limit(calls = 5, period = 1) %%
  function(url) url
```

 retry

Retry a function when it fails

Description

Decorator that calls a function again until it succeeds or a [stop condition](#) condition is met, sleeping between attempts. Port of Python's tenacity.

Usage

```
retry(
    f,
    stop = stop_never(),
    wait = wait_none(),
    retry = retry_all(),
    before_sleep = before_sleep_none(),
    after_sleep = after_sleep_none(),
    reraise = FALSE,
    retry_error_callback = NULL
)
```

Arguments

<code>f</code>	The function to decorate. Usually supplied by <code>%%</code> rather than directly.
<code>stop</code>	A stop condition, e.g. stop_after_attempt() . Defaults to never stopping.
<code>wait</code>	A wait strategy . Defaults to no waiting.
<code>retry</code>	A retry condition . Defaults to retrying every error.
<code>before_sleep, after_sleep</code>	Hooks run before and after each sleep, see before_sleep_log() and after_sleep_log() .
<code>reraise</code>	If TRUE, re-throw the last error when stop fires instead of throwing a <code>retry_error</code> .
<code>retry_error_callback</code>	Optional function called with the final state when stop fires. Its return value is returned instead of throwing. The state is a list with <code>attempt</code> , <code>wait</code> , <code>elapsed</code> , <code>error</code> and <code>value</code> .

Details

Call `retry(stop, wait, retry, before_sleep, after_sleep, reraise, retry_error_callback)` to build a decorator, then apply it with `%%` or by calling it on a function.

Value

A decorator. The decorated function returns the first acceptable result, or signals a `retry_error` condition (with `$attempt`, `$error` and `$value`) once stop fires.

Examples

```

n <- 0
flaky <- function() {
  n <- n + 1
  if (n < 3) stop("not yet")
  n
}
retry(stop = stop_after_attempt(5))(flaky)()

# Back off exponentially between attempts
f <- retry(stop = stop_after_attempt(3), wait = wait_exp(min = 0.1)) %% function() 1
f()

```

 retry_all

Retry conditions

Description

Decide whether `retry()` tries again after an attempt.

Usage

```

retry_all()

retry_never()

retry_if_error(classes = "error")

retry_if_error_message(message = NULL, match = NULL)

retry_if_result(predicate = function() NULL)

```

Arguments

<code>classes</code>	Condition classes to retry on.
<code>message</code>	Error message to match exactly.
<code>match</code>	Regular expression to match against the error message.
<code>predicate</code>	Function taking the returned value and returning TRUE to retry.

Details

- `retry_all()` retries on any error.
- `retry_never()` never retries, so the first error is re-thrown.
- `retry_if_error()` retries on errors inheriting from any of classes.
- `retry_if_error_message()` retries when the error message equals `message`, or matches the regular expression `match`. Supply exactly one.
- `retry_if_result()` retries when the call succeeds but `predicate(value)` is TRUE.

Value

A retry condition to pass to `retry()`.

Examples

```
retry_if_error("http_error")
retry_if_error_message(match = "timed? ?out")
retry_if_result(is.null)
```

stop_never	<i>Stop conditions</i>
------------	------------------------

Description

Decide when `retry()` gives up. A condition is checked after each failed attempt.

Usage

```
stop_never()

stop_after_attempt(max_attempt_number = 1)

stop_after_delay(max_delay = 1)

stop_before_delay(max_delay = 1)

stop_any(...)

stop_all(...)
```

Arguments

```
max_attempt_number    Number of attempts to allow.
max_delay              Seconds since the first attempt.
...                   Stop conditions to combine.
```

Details

- `stop_never()` never stops.
- `stop_after_attempt()` stops after a number of attempts.
- `stop_after_delay()` stops once that many seconds have passed since the first attempt.
- `stop_before_delay()` stops if sleeping again would reach that many seconds.
- `stop_any()` stops when any of the given conditions is met.
- `stop_all()` stops when all of the given conditions are met.

Value

A stop condition to pass to `retry()`.

Examples

```
stop_after_attempt(3)
stop_any(stop_after_attempt(5), stop_after_delay(10))
```

validate_args	<i>Validate arguments and return values</i>
---------------	---

Description

Decorators that check a function's inputs or output against validators you supply. A validator is any function (or `rlang::as_function()` formula) that takes the value, so base predicates, lambdas and the `check_*()` functions of packages like `checkmate` all work, with no dependency on any of them.

Usage

```
validate_args(...)

validate_return(validator)
```

Arguments

<code>...</code>	For <code>validate_args()</code> , named validators, one per argument.
<code>validator</code>	For <code>validate_return()</code> , a validator for the returned value.

Details

A validator passes if it returns exactly `TRUE`. It fails if it returns anything else:

- a string, which becomes the failure message (the `check_*()` convention),
- `FALSE` or any other value, which gives a generic message.

If a validator signals an error, that error propagates unchanged, so `assert_*()` style functions work too.

`validate_args()` checks the effective value of each named argument, so defaults are validated too. Checking forces the argument, so a validated argument is evaluated at call time rather than lazily. All failures are reported in one error of class `garnish_validation_error`. Validator names are checked against the function's formals when it is decorated, so a typo fails early.

Value

A decorator.

Examples

```
f <- validate_args(x = is.numeric, n = ~ .x > 0) %% function(x, n = 1) x * n
f(2, 3)
try(f("a", -1))

# check_*() style validators return a message on failure
check_string <- function(x) if (is.character(x) && length(x) == 1) TRUE else "must be a string"
g <- validate_args(name = check_string) %% function(name) name
try(g(1))

h <- validate_return(is.character) %% function() 1
try(h())
```

wait_fixed

*Wait strategies***Description**

Decide how many seconds `retry()` sleeps after a failed attempt.

Usage

```
wait_fixed(wait = integer(0))

wait_none()

wait_exp(multiplier = 1, min = 0, max = 4611686018427387904, exp_base = 2)

wait_exp_jitter(base = 0.1, max_delay = 20, jitter = "full")

wait_rand(min = 0, max = 1)

wait_chain(...)

wait_combine(...)
```

Arguments

<code>wait</code>	Seconds to wait.
<code>multiplier, exp_base</code>	Scale and base of the exponential.
<code>min, max</code>	Lower and upper bound in seconds.
<code>base</code>	Seconds to wait on the first attempt.
<code>max_delay</code>	Longest wait in seconds.
<code>jitter</code>	Jitter type, one of "full", "equal", "decorrelated" or "none".
<code>...</code>	Wait strategies to chain or combine.

Details

- `wait_none()` does not wait.
- `wait_fixed()` waits a fixed time.
- `wait_rand()` waits a random time between min and max.
- `wait_exp()` waits `multiplier * exp_base^(attempt - 1)`, kept between min and max.
- `wait_exp_jitter()` waits `base * 2^(attempt - 1)`, capped at `max_delay`, then applies jitter: "none", "full" (uniform from 0), "equal" (half fixed, half random) or "decorrelated" (random between base and 3 times the previous wait).
- `wait_chain()` uses the n-th strategy for the n-th attempt, then repeats the last.
- `wait_combine()` adds strategies together. `a + b` is the same as `wait_combine(a, b)`.

Value

A wait strategy to pass to `retry()`.

Examples

```
wait_fixed(2)
wait_exp(multiplier = 0.5, max = 30)
wait_chain(wait_fixed(1), wait_fixed(5))
wait_fixed(1) + wait_rand(0, 2)
```

with_context

Run code inside a context

Description

The scoped form of a `contextmanager()`. Evaluates `ctx` with cleanup tied to this call, passes the value it returns to code, and cleans up when code finishes, even if it errors.

Usage

```
with_context(ctx, code)
```

Arguments

<code>ctx</code>	A call to a context manager, e.g. <code>local_tempdir()</code> . It is not evaluated in your frame: <code>.local_envir</code> is set to this function's frame.
<code>code</code>	A function taking the context's value.

Value

The value of code.

Examples

```
local_tempdir := contextmanager %% function() {  
  path <- tempfile()  
  dir.create(path)  
  teardown(unlink(path, recursive = TRUE))  
  path  
}  
with_context(local_tempdir(), function(d) dir.exists(d))
```

Index

after_sleep_base, 2
after_sleep_fn(after_sleep_base), 2
after_sleep_log(after_sleep_none), 4
after_sleep_log(), 11
after_sleep_none, 4
as_decorator, 5
as_decorator(), 7, 8

before_sleep_base(after_sleep_base), 2
before_sleep_fn(after_sleep_base), 2
before_sleep_log(before_sleep_none), 5
before_sleep_log(), 11
before_sleep_none, 5

cache(lru_cache), 8
circuit_breaker, 6
contextmanager, 7
contextmanager(), 5, 16

disk_cache(lru_cache), 8

extending_retry(after_sleep_base), 2

lru_cache, 8

message(), 4, 6

rate_limit, 10
retry, 11
retry condition, 11
retry(), 2, 4–6, 10, 12–16
retry_all, 12
retry_base(after_sleep_base), 2
retry_fn(after_sleep_base), 2
retry_if_error(retry_all), 12
retry_if_error_message(retry_all), 12
retry_if_result(retry_all), 12
retry_never(retry_all), 12
rlang::as_function(), 14

S7::methods_register(), 3

saveRDS(), 9
stop condition, 11
stop_after_attempt(stop_never), 13
stop_after_attempt(), 11
stop_after_delay(stop_never), 13
stop_all(stop_never), 13
stop_all(), 3
stop_any(stop_never), 13
stop_any(), 3
stop_base(after_sleep_base), 2
stop_before_delay(stop_never), 13
stop_fn(after_sleep_base), 2
stop_never, 13

teardown(contextmanager), 7

validate_args, 14
validate_return(validate_args), 14

wait strategy, 11
wait_base(after_sleep_base), 2
wait_chain(wait_fixed), 15
wait_chain(), 3
wait_combine(wait_fixed), 15
wait_exp(wait_fixed), 15
wait_exp_jitter(wait_fixed), 15
wait_fixed, 15
wait_fn(after_sleep_base), 2
wait_none(wait_fixed), 15
wait_rand(wait_fixed), 15
with_context, 16
with_context(), 7, 8
withr::defer(), 8