

Package ‘nofrills’

January 8, 2021

Type Package

Title Low-Cost Anonymous Functions

Version 0.3.1

Description Provides a compact variation of the usual syntax of function declaration, in order to support tidyverse-style quasiquotation of a function's arguments and body.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

ByteCompile true

Depends R (>= 3.1.0)

Imports rlang (>= 0.2.0)

Suggests testthat, dplyr (>= 0.7.0)

URL <https://github.com/egnha/nofrills>

BugReports <https://github.com/egnha/nofrills/issues>

Collate 'nofrills.R' 'closure.R' 'exprs.R' 'fn.R' 'as-fn.R'
'make-fn-aware.R' 'curry.R'

RoxygenNote 7.1.0

NeedsCompilation no

Author Eugene Ha [aut, cre]

Maintainer Eugene Ha <eha@posteo.de>

Repository CRAN

Date/Publication 2021-01-08 19:50:05 UTC

R topics documented:

as_fn	2
curry	3
fn	4
make_fn_aware	8

as_fn	<i>Abbreviated functional arguments</i>
-------	---

Description

as_fn() is for functions that take functional arguments. Use as_fn() *inside* a function to enable it to comprehend a minimal anonymous-function notation for arguments that are functions. This notation is that of fn(), but with 'fn' replaced by '.' (dot).

Usage

```
as_fn(.f)
```

Arguments

.f	A function or an abbreviated anonymous-function expression of the form .(...), where ... is a function declaration (i.e., . (dot) in this context is an alias of fn()). Quasiquotation is supported.
----	--

Details

as_fn() cannot follow promise expressions across function calls. It is only intended to work in the immediate context in which a function declaration is to be interpreted (see *Examples*).

Value

If .f is a function, it is simply returned, otherwise the function determined by the [function declaration](#) is returned.

See Also

[fn\(\)](#), [make_fn_aware\(\)](#)

Examples

```
call_fn <- function(.f, x) {
  f <- as_fn(.f)
  f(x)
}
call_fn(log, 1)
call_fn(.(. ~ sin(.) ^ 2), 1)
# simplified function expressions support quasiquotation
f <- sin
call_fn(.(. ~ (!!f)(.) ^ 2), 1)

## wrap Map() to accept abbreviated anonymous function expressions
Map_ <- function (f, ...) {
  f <- as_fn(f)
```

```

  mapply(FUN = f, ..., SIMPLIFY = FALSE)
}
# you can call Map_() just like Map()
Map_(function(x, y, z) paste(x, y, paste("and", z), sep = ", "), 1:3, 4:6, 7:9)
# or use a simplified function expression
Map_.(x, y, z ~ paste(x, y, paste("and", z), sep = ", "), 1:3, 4:6, 7:9)

## abbreviated anonymous functions are interpreted in the calling environment
# so this works, as expected
foo <- function(a) as_fn(a)
foo.(x ~ x + 1)
# but as_fn() can't interpret abbreviated anonymous functions across calls
foo <- function(a) bar(a)
bar <- function(b) as_fn(b)
## Not run:
foo.(x ~ x + 1)
## End(Not run)

```

curry

Curry a function

Description

`curry()` **curries** functions—it reconstitutes a function as a succession of single-argument functions. For example, `curry()` produces the the function

```

function(x) {
  function(y) {
    function(z) {
      x * y * z
    }
  }
}

```

from the function `function(x,y,z) x * y * z`.

`curry_fn()` produces a curried function from an `fn()`-style function declaration, which supports [quasiotation](#) of a function's body and (default) argument values.

Usage

```
curry(f, env = environment(f))
```

```
curry_fn(..., ..env = parent.frame())
```

Arguments

f	Function.
env	Environment of the curried function or NULL. If NULL, the environment of the curried function is the calling environment.
...	Function declaration, which supports quasiquote .
..env	Environment in which to create the function (i.e., the function's enclosing environment).

Details

Dots (...) are treated as a unit when currying. For example, `curry()` transforms `function(x, ...) list(x, ...)` to `function(x) { function(...) list(x, ...) }`.

Value

A function of nested single-argument functions.

See Also

[fn\(\)](#)

Examples

```
curry(function(x, y, z = 0) x + y + z)
double <- curry(`*`)(2)
double(3) # 6

curry_fn(x, y, z = 0 ~ x + y + z)
curry_fn(target, ... ~ identical(target, ...))

## Delay unquoting to embed argument values into the innermost function
compare_to <- curry_fn(target, x ~ identical(x, QUQ(target)))
is_this <- compare_to("this")
is_this("that") # FALSE
is_this("this") # TRUE
classify_as <- curry_fn(class, x ~ `class<-`(x, QUQ(class)))
as_this <- classify_as("this")
as_this("Some object") # String of class "this"
```

Description

`fn()` enables you to create (anonymous) functions, of arbitrary call signature. Use it in place of the usual `function()` invocation whenever you want to:

- type less:

```
fn(x, y = 1 ~ x + y)
function(x, y = 1) x + y
```

are equivalent

- guard against changes in lexical scope: by enabling tidyverse [quasiquote](#), `fn()` allows you to “burn in” values at the point of function creation (see *Pure functions via quasiquote*)

Usage

```
fn(..., ..env = parent.frame())
```

Arguments

<code>...</code>	Function declaration, which supports quasiquote .
<code>..env</code>	Environment in which to create the function (i.e., the function’s enclosing environment).

Value

A function whose enclosing environment is `..env`.

Function declarations

A *function declaration* is an expression that specifies a function’s arguments and body, as a comma-separated expression of the form

```
arg1, arg2, ..., argN ~ body
```

or

```
arg1, arg2, ..., argN, ~ body
```

(Note in the second form that the body is a one-sided formula. This distinction is relevant for argument [splicing](#), see below.)

- To the left of `~`, you write a conventional function-argument declaration, just as in `function(<arguments>)`: each of `arg1`, `arg2`, ..., `argN` is either a bare argument (e.g., `x` or `...`) or an argument with default value (e.g., `x = 1`).
- To the right of `~`, you write the function body, i.e., an expression of the arguments.

Quasiquote: All parts of a function declaration support tidyverse [quasiquote](#):

- To unquote values (of arguments or parts of the body), use `!!`:

```
z <- 0
fn(x, y = !!z ~ x + y)
fn(x ~ x > !!z)
```

- To unquote argument names (with default value), use := (definition operator):

```
arg <- "y"
fn(x, !!arg := 0 ~ x + !!as.name(arg))
```

- To splice in a (formal) list of arguments, use !!!:

```
fn(!!!alist(x, y = 0), ~ x + y)
```

(Note that the body, in this case, must be given as a one-sided formula.)

- To write literal unquoting operators, use QUQ(), QUQS():

```
library(dplyr)
```

```
my_summarise <- fn(df, ... ~ {
  group_by <- quos(...)
  df %>%
    group_by(QUQS(group_by)) %>%
    summarise(a = mean(a))
})
```

(Source: *Programming with dplyr*)

Pure functions via quasiquotation

Functions in R are generally **impure**, i.e., the return value of a function will *not* in general be determined by the value of its inputs alone. This is because a function may depend on mutable objects in its **lexical scope**. Normally this isn't an issue. But if you are working interactively and sourcing files into the global environment, say, or using a notebook interface (like Jupyter or R Notebook), it can be tricky to ensure that you haven't unwittingly mutated an object that an earlier function depends upon.

Example — Consider the following function:

```
a <- 1
foo <- function(x) x + a
```

What is the value of `foo(1)`? It is not necessarily 2, because the value of `a` may have changed between the *creation* of `foo()` and the *calling* of `foo(1)`:

```
foo(1) #> [1] 2
a <- 0
foo(1) #> [1] 1
```

In other words, `foo()` is impure because the value of `foo(x)` depends not only on the value of `x` but also on the *externally mutable* value of `a`.

`fn()` enables you to write *pure* functions by using **quasiquotation** to eliminate such indeterminacy.

Example — With `fn()`, you can unquote `a` to “burn in” its value at the point of creation:

```
a <- 1
foo <- fn(x ~ x + !!a)
```

Now `foo()` is a pure function, unaffected by changes in its lexical scope:

```
foo(1) #> [1] 2
a <- 0
foo(1) #> [1] 2
```

See Also

[as_fn\(\)](#), [make_fn_aware\(\)](#), [curry_fn\(\)](#)

Examples

```
fn(x ~ x + 1)
fn(x, y ~ x + y)
fn(x, y = 2 ~ x + y)
fn(x, y = 1, ... ~ log(x + y, ...))

## to specify '...' in the middle, write '... = '
fn(x, ... = , y ~ log(x + y, ...))

## use one-sided formula for constant functions or commands
fn(~ NA)
fn(~ message("!"))

## unquoting is supported (using `!!` from rlang)
zero <- 0
fn(x = !!zero ~ x > !!zero)

## formals and function bodies can also be spliced in
f <- function(x, y) x + y
g <- function(y, x, ...) x - y
frankenstein <- fn(!!!formals(f), ~ !!body(g))
stopifnot(identical(frankenstein, function(x, y) x - y))

## mixing unquoting and literal unquoting is possible
if (suppressWarnings(require(dplyr))) {
  summariser <- quote(mean)

  my_summarise <- fn(df, ... ~ {
    group_by <- quos(...)
    df %>%
      group_by(QUQS(group_by)) %>%
      summarise(a = `!!`(summariser)(a))
  })

  my_summarise
}
```

`make_fn_aware`*Make a function aware of abbreviated functional arguments*

Description

`make_fn_aware()` is a functional operator that enhances a function by enabling it to interpret abbreviated functional arguments.

Usage

```
make_fn_aware(f, ...)
```

Arguments

<code>f</code>	Function, or symbol or name of a function.
<code>...</code>	Name(s) of functional argument(s) of <code>f</code> (strings) or NULL. Unsplicing of lists of strings is supported via <code>!!!</code> .

Value

A function with the same call signature as `f`, but whose function arguments, as designated by `...`, may be specified using an abbreviated function expression of the form `.(...)`, cf. [as_fn\(\)](#). If `...` is empty or NULL, then `f` is simply returned.

See Also

[as_fn\(\)](#)

Examples

```
reduce <- make_fn_aware(Reduce, "f")

## reduce() behaves just like Reduce()
Reduce(function(u, v) u + 1 / v, c(3, 7, 15, 1, 292), right = TRUE)
reduce(function(u, v) u + 1 / v, c(3, 7, 15, 1, 292), right = TRUE)

## reduce() can also interpret abbreviated function expressions
reduce.(u, v ~ u + 1 / v), c(3, 7, 15, 1, 292), right = TRUE)
```


Index

`as_fn`, 2

`as_fn()`, 7, 8

`curry`, 3

`curry_fn (curry)`, 3

`curry_fn()`, 7

enclosing environment, 4, 5

`fn`, 4

`fn()`, 2–4

function declaration, 2

`make_fn_aware`, 8

`make_fn_aware()`, 2, 7

Quasiquotation, 2

quasiquotation, 3–5

splicing, 5