Package: svFlow (via r-universe)

August 29, 2024

Type Package

Version 1.2.1

Title Data Analysis Work Flow and Pipeline Operator for 'SciViews::R'

Description Data work flow analysis using 'proto' objects and pipe operator that integrates non-standard evaluation and the 'lazyeval' mechanism.

Maintainer Philippe Grosjean <phgrosjean@sciviews.org>

Depends R (>= 4.2.0)

Imports graphics (>= 4.2.0), igraph (>= 1.4.2), proto (>= 1.0.0), rlang (>= 0.2.0), utils (>= 4.2.0)

Suggests datasets (>= 4.2.0), dplyr (>= 1.1.4), microbenchmark (>= 1.4.9), knitr (>= 1.42), rmarkdown (>= 2.21), spelling (>= 2.2.1), testthat (>= 3.0.0), lintr (>= 3.0.2)

License MIT + file LICENSE

URL https://github.com/SciViews/svFlow, https://www.sciviews.org/SciViews-R/

BugReports https://github.com/SciViews/svFlow/issues

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

VignetteBuilder knitr

Encoding UTF-8

Language en-US

ByteCompile yes

Config/testthat/edition 3

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

RemoteUrl https://github.com/SciViews/svFlow

RemoteRef HEAD

RemoteSha e827bf4b5adff5b400cb48eef1635d1510a6c316

Contents

`svFlow-package` *Data Analysis Work Flow and Pipeline Operator for 'SciViews::R'*

Description

Data (work)flow analysis using proto objects (see `proto()`) and a pipe operator that integrates non-standard evaluation and the tidyeval mechanism in a most transparent way.

Important functions

- `%>.%` and `%>_%` are two alternate pipe operators designed to supplement magrittr's \ tidyverse and elsewhere. They are provided for good reasons. `%>.%` requires explicit indication of the position of . in the pipeline expression **all the time**. The expression is not modified. As a consequence, it can never surprise you with an unexpected behavior, and all valid R expressions are usable in the pipeline. Another consequence: it is very fast. `%>_%` works with **Flow** objects that allow for encapsulation of satellite objects (data or functions) within the pipeline. It is self-contained. The pipeline can be interrupted and restarted at any time. It also allows for a class-less object-oriented approach with single inheritance (could be useful to test easily different scenarios on the same pipeline and to prototype objects that are "pipe-aware"). It also manages the tidyeval mechanism for non-standard expressions in the most transparent way: the only "rule" to remember is to suffix the name of variables that needs special treatment with an underscore (_) and the pipe operator manages the rest for you.
- `debug_flow()` provides a convenient way to debug problematic pipelines build with our own pipe operators `%>.%` and `%>_%` in a comfortable way. Everything from the step that raised a error is available: the piped data, the expression to be evaluated, and possibly, the last state of the **Flow** object. Everything can be inspected, modified, and the expression can be rerun as if you were still right in the middle of the pipeline evaluation.
- `flow()` constructs a `Flow` object that is pipe-aware and tidyeval-aware. This opens new horizons in your analysis workflow. You start building a simple *ad hoc* pipeline, then you can include satellite data or functions right inside it, perhaps also test different scenarios by using the object inheritance features of **Flow** (common parts are shared among the different scenarios, thus reducing the memory footprint). While your pipeline matures you gradually and naturally move towards either a functional sequence or a dedicated object. The functional sequence pathway consists in building a reusable function to recycle you pipeline in a different context. The object pathway is not fully developed yet in the present version. But in

the future, the object-oriented nature of **Flow** will also be leveraged, so that you could automatically translate your "flow pipeline" into an S3 or R6 object with satellite data becoming object attributes, and satellite functions becoming methods. The pipeline itself would then become the default method for that object. Of course, both functions and objects derived from a "flow pipeline" will be directly compatible with the tidyeval mechanism, as they will be most tidyverse-friendly as possible per construction.

- `str.Flow()` compactly displays the content of a **Flow** object.
- `as.quosure()`, and unary + and - operators combined with **formula** objects provide an alternate way to create **quosure**s.
- `quos_underscore()` automatically converts arguments whose name ends with _ into **quosure**s, and this mechanism is used by our flow pipe operator to implement the tidyeval mechanism most transparently inside "flow pipelines".

`._` *Pass first argument as dot to run code in second argument for pipe operators that do not natively support dot-replacement scheme (base R pipe operator)*

Description

Pass first argument as dot to run code in second argument for pipe operators that do not natively support dot-replacement scheme (base R pipe operator)

Usage

```
._(x, expr)
```

Arguments

`x`	Object to pass to `expr` as dot (`.`).
`expr`	Expression to execute, containing `.` as a placeholder.

Details

The function has a side-effect to assign `x` as `.` and unevaluated `expr` as `.call` in the calling environment. Therefore, make sure you do not use `.` or `.call` there for something else. In case `expr` fails in the middle of a series of chained pipes, you can inspect `.` and `.call` or possibly rerun a modified version of the instruction that failed on it for easier debugging purpose.

Value

The result from executing `expr` in the parent environment.

Examples

```
# The function is really supposed to be use in a pipe instruction
# This example only runs on R >= 4.1
## Not run:
# lm has data = as second argument, which does not fit well with the pipe |>
# In R 4.1, one should write:
iris |> \(.)(lm(data = ., Sepal.Length ~ Petal.Length + Species))()
# which is not very elegant ! With ._() it is more concise and straighforward
iris |> ._(lm(data = ., Sepal.Length ~ Petal.Length + Species))

## End(Not run)
```

`flow` *Create Flow objects to better organize pipelines in R*

Description

Flow objects, as explicitly created by `flow()`, or implicitly by the `%>_%` pipe operator are **proto** objects (class-less objects with possible inheritance) that can be combined nicely with pipelines using the specialized flow pipe operator `%>_%` (or by using `$`). They allow for encapsulating satellite objects/variables related to the pipeline, and they deal with non-standard evaluations using the tidyeval mechanism automatically with minimal changes required by the user.

Usage

```
flow(. = NULL, .value = NULL, ...)

enflow(.value, env = caller_env(), objects = ls(env))

is.flow(x)

is_flow(x)

as.flow(x, ...)

as_flow(x, ...)

## S3 method for class 'Flow'
x$name

## S3 replacement method for class 'Flow'
x$name <- value
```

Arguments

.	If a **Flow** object is provided, inherit from it, otherwise, create a new **Flow** object inheriting from `.GlobalEnv` with `.` as pipe value.

`.value`	The pipe value to pass to the object (used instead of `.`, in case both are provided).
`...`	For `flow()`, named arguments of other objects to create inside the **Flow** object. If the name ends with _, then, the expression is automatically captured inside a *quosure** (see `quos_underscore()`). For `print()`, further arguments passed to the delegated `object_print()` function (if it exists inside the **Flow** object), or to the `print()` method of the object inside `.value`.
`env`	The environment to use for populating the **Flow** object. All objects from this environment are injected into it, with the objects not starting with a dot and ending with an underscore (_) automatically converted into `quosures`. The object provided to `.value=` becomes the default value of the `Flow` object, that is, the data transferred to the pipeline.
`objects`	A character string with the name of the objects from `env` to import into the **Flow** object. If `env` is the calling environment (by default), `.value` is the name of an object, and that name appears in `objects` too, it is excluded from it to avoid importing it twice. from that
`x`	An object (a **Flow** object, or anything to test if it is a **Flow** object in `is_flow()`).
`name`	The name of the item to get from a **Flow** object. If `name` starts with two dots (`..`), the item is searched in the **Flow** object itself without inheritance, but the name is stripped of its leading two dots first! If the content is a **quosure**, it is automatically unquoted, and for the assignation version, if name ends with _, the expression is automatically converted into a **quosure**.
`value`	The value or expression to assign to `name` inside the **Flow** object.

Details

`enflow()` creates a **Flow** object in the head of a "flow pipeline" in the context of a functional sequence, that is a function that converts an *ad hoc*, single use pipeline into a function reusable in a different context. Satellite data become arguments of the function.

When a **Flow** object is created from scratch, it always inherits from `.GlobalEnv`, no matter where the expression was executed (in fact, it inherits from an empty root **Flow** object itself inheriting from `.GlobalEnv`). This is a deliberate design choice to overcome some difficulties and limitations of **proto** objects, see `proto()`. `enflow()` creates a **Flow** object and populates it automatically with all the objects that are present in `env=` (by default, the calling environment). It is primarily intended to be used inside a function, as first instruction of a "flow pipeline". Hence, it collects all function arguments inside that pipeline in a most convenient way.

See Also

str.Flow, quos_underscore, %>_%

Examples

```
library(svFlow)
library(dplyr)
data(iris)

foo <- function(data, x_ = Sepal.Length, y_ = log_SL,
```

```
fun_ = mean, na_rm = TRUE)
  enflow(data) %>_%
  mutate(., y_ = log(x_)) %>_%
  summarise(., fun_ = fun_(y_,
    na.rm = na_rm_)) %>_% .

foo(iris)

foo(iris, x_ = Petal.Width)

foo(iris, x_ = Petal.Width, fun_ = median)
# Unfortunately, this does not work, due to limitations of tidyeval's :=
#foo(iris, x_ = Petal.Width, fun_ = stats::median)

foo2 <- function(., x_ = Sepal.Length, y_ = log_SL, na_rm = TRUE)
  enflow(.)

foo2
foo2(1:10) -> foo_obj
ls(foo_obj)
```

`graph_flow` *Create a graph with Flow objects hierarchy*

Description

A graph showing all **Flow** objects heritage is calculated, and displayed.

Usage

```
graph_flow(env = .GlobalEnv, child_to_parent = TRUE, plotit = TRUE, ...)
```

Arguments

`env`	The environment to look for **Flow** objects. By default, it is `.GlobalEnv`, and you should not change it, since all **Flow** objects are derived from it by construction.
`child_to_parent`	Do the arrows go from child to parent (by default), or in the other direction?
`plotit`	Do we plot the graph (by default)?
`...`	Further parameters passed to `plot.igraph()`.

Value

An **igraph** object (returned invisibly if `plotit = TRUE`.

See Also

flow

Examples

```
a <- flow()
b <- a$flow()
c <- b$flow()
d <- a$flow()
# Use of custom names
e <- flow(.name = "parent")
f <- e$flow(.name = "child")
graph_flow()

# Arrows pointing from childs to parents, and do not plot it
g <- graph_flow(child_to_parent = FALSE, plotit = FALSE)
g
plot(g)
```

`pipe_operator` *Flow pipeline operators and debugging function*

Description

Pipe operators. `%>.%` is a very simple and efficient pipe operator. `%>_%` is more complex. It forces conversion to a **Flow** object inside a pipeline and automatically manage non-standard evaluation through creation and unquoting of **quosure**s for named arguments whose name ends with _.

Usage

```
x %>.% expr

x %>_% expr

debug_flow()
```

Arguments

`x`	Value or **Flow** object to pass to the pipeline.
`expr`	Expression to evaluation in the pipeline.

Details

With `%>.%`, the value must be explicitly indicated with a `.` inside the expression. The expression is **not** modified, but the value is first assigned into the calling environment as `.` (warning! possibly replacing any existing value... do **not** use `.` to name other objects). Also the expression is saved as `.call` in the calling environment so that `debug_flow()` can retrieve are rerun it easily. If a **Flow** object is used with `%>.%`, the `.value` is extracted from it into `.` first (and thus the **Flow** object is lost).

In the case of `%>_%` the **Flow** object is passed or created, it is also assigned in the calling environment as `...` This can be used to refer to **Flow** object content within the pipeline expressions (e.g., `..$var`).

For %>_%, the expression is reworked in such a way that a suitable lazyeval syntax is constructed for each variable whose name ends with _, and that variable is explicitly searched starting from ... Thus, x_ is replaced by !!..$x. For such variables appearing at left of an = sign, it is also replaced by := to keep correct R syntax (var_ = => !!..$var :=). This way, you just need to follow special variables by _, both in the flow() function arguments (to create quosures), and to the NSE expressions used inside the pipeline to get the job done! The raw expression is saved as .call_raw, while the reworked call is saved as .call for possible further inspection and debugging.

Finally, for %>_%, if expr is ., then, the last value from the pipe is extracted from the **Flow** object and returned. It is equivalent, thus, to flow_obj$.value.

You can mix %>.% and %>_% within the same pipeline. In case you use %>.% with a flow pipeline, it "unflows" it, extracting .value from the **Flow** object and further feeding it to the pipeline.

See Also

flow, quos_underscore

Examples

```
# A simple pipeline with %>.% (explicit position of '.' required)
library(svFlow)
library(dplyr)
data(iris)
iris2 <- iris %>.%
  mutate(., log_SL = log(Sepal.Length)) %>.%
  filter(., Species == "setosa")

# The %>.% operator is much faster than magrittr's %>%
# (although this has no noticeable impact in most situations when the
# pipeline in used in an ad hoc way, outside of loops or other constructs
# that call it a larger number of times)
```

---

quosure *Create and manipulate quosures easily*

Description

Quosures are defined in {rlang} package as part of the tidy evaluation of non-standard expressions (see quo()). Here, we provide an alternate mechanism using -~expr as a synonym of quo(expr). Also, +quo_obj is equivalent to !!quo_obj in {rlang}, and ++quo_obj both unquotes and evaluates it in the right environment. Quosures are keystone objects in the tidy evaluation mechanism. So, they deserve a special, clean and concise syntax to create and manipulate them.

The as_xxx() and is_xxx() further ease the manipulation of **quosure**s or related objects.

Usage

```
## S3 method for class 'formula'
e1 - e2

## S3 method for class 'formula'
e1 + e2

## S3 method for class 'quosure'
e1 ^ e2

## S3 method for class 'quosure'
e1 + e2

## S3 method for class 'unquoted'
e1 + e2

## S3 method for class 'unquoted'
print(x, ...)

as.quosure(x, env = caller_env())

is.quosure(x)

is.formula(x)

is.bare_formula(x)

`!!`(x)
```

Arguments

e1	Unary operator member, or first member of a binary operator.
e2	Second member of a binary operator (not used here, except for ^).
x	An expression
...	Further arguments passed to the print() method (not used yet).
env	An environment specified for scoping of the quosure.

Details

- is defined as an unary minus operator for **formula** objects (which is *not* defined in base R, hence, not supposed to be used otherwise). Thus, -~expr just converts a formula build using the base ~expr instruction into a **quosure**. as.quosure() does the same, when the expression is provided directly, and allows also to define the enclosing environment (by default, it is the environment where the code is evaluated, and it is also the case when using -~expr).

Similarly, the unary + operator is defined for **quosure** in order to easily "reverse" the mechanism of quoting an expression with a logical complementary operator. It does something similar to !! in {rlang}, but it can be used outside of tidy eval expressions. Since unary + has higher syntax

precedence than ! in R, it is less susceptible to require parentheses (only ^ for exponentiation, indexing/subsetting operators like $ or [, and namespace operators :: and ::: have higher precedence). A specific ^ operator for quosures solves the precedence issue. :: or ::: are very unlikely used in the context.

++quosure is indeed a two-steps operation (+(+quosure)). It first unquotes the quosure, returning an **unquoted** object. Then, the second + evaluates the **unquoted** object. This allows for fine-graded manipulation of **quosure**s: you can unquote at one place, and evaluate the **unquoted** object elsewhere (and, of course, the contained expression is always evaluated in the *right* environment, despite all these manipulations).
!! and just evaluates its argument and passes the result. It is only useful inside a quasi-quoted argument, see quasiquotation.

Value

These functions build or manipulated **quosure**s and return such objects. +quosure creates an **unquoted** object. The + unary operator applied to **unquoted** objects evaluate the expression contained in the **quosure** in the right environment.

See Also

quos_underscore, %>_%

Examples

```
x <- 1:10
# Create a quosure (same as quo(x))
x_quo <- ~~x
x_quo
# Unquote it (same as !!x, but usable everywhere)
+x_quo
# Unquote and evaluate the quosure
++x_quo
# Syntax precedence issues (^ has higher precedence than unary +)
# is solved by redefining ^ for unquoted objects:
++x_quo^2
# acts like if ++ had higher precedence than ^, thus like if it was
(++x_quo)^2

# Assign the unquoted expression
x_unquo <- +x_quo
# ... and use x_unquo in a different context
foo <- function(x) +x
foo(x_unquo)
```

---

quos_underscore *Convert arguments whose names end with _ into quosures automatically*

Description

The expressions provided for all arguments whose names end with _ are automatically converted into **quosure**s, and also assigned to a name without the training _. The other arguments are evaluated in an usual way.

Usage

```
quos_underscore(...)
```

Arguments

...	The named arguments provided to be either converted into quosures or evaluated.

Value

An object of class **quosures** is returned. It can be used directly in tidyeval-aware contexts.

See Also

as.quosure, %>_%

Examples

```
foo <- function(...)
  quos_underscore(...)
foo(x  = 1:10, # "Normal" argument
    y_ = 1:10, # Transformed into a quosure
    z_ = non_existing_name) # Expressions in quosures are not evaluated
```

---

str.Flow *Compactly display the content of a Flow object*

Description

Print short informative strings about the **Flow** object and all it contains, plus possibly, inheritance information.

Usage

```
## S3 method for class 'Flow'
str(
  object,
  max.level = 1L,
  nest.lev = 0L,
  indent.str = paste(rep.int(" ", max(0L, nest.lev + 1L)), collapse = ".."),
  ...
)
```

Arguments

`object`	A **Flow** object.
`max.level`	The maximum nesting level to use for displaying nested structures.
`nest.lev`	Used internally for pretty printing nested objects (you probably don't want to change default value).
`indent.str`	Idem.
`...`	Further arguments passed to `str()` methods of **Flow** items.

See Also

flow

Examples

```
# A Flow object
data(iris)
fl <- flow(iris, x = 1:10, var_ = Sepal.Length)
fl # Shows the .value contained into fl
str(fl) # Provides compact information about satellite data contained in fl
```

Index