



Modernization of Statistical Production with R

Mark van der Loo

Statistics Netherlands, University of Leiden

2023-12-13

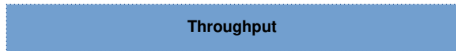
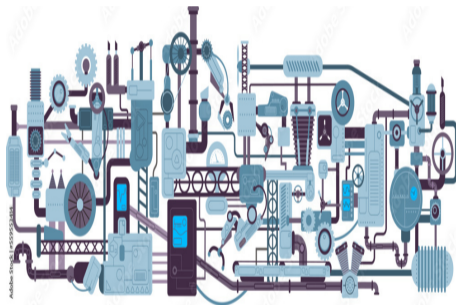
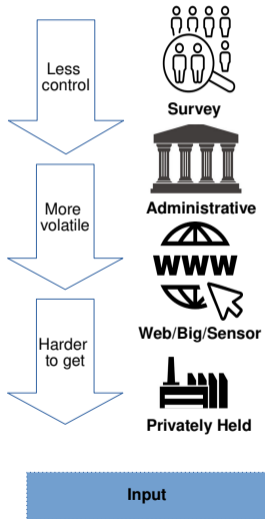




Modernization



General Motivation



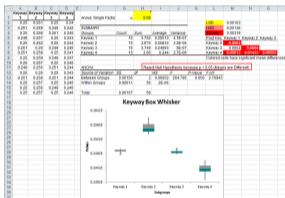
Legal obligations
Comparable over time and space



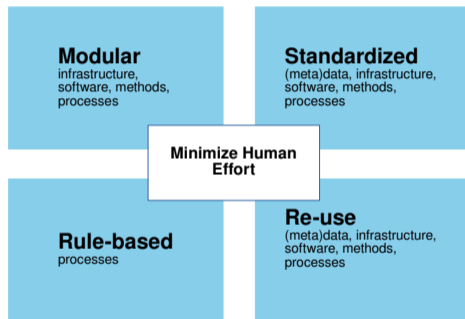
Current events
Quick Response



Modernization at Statistics Netherlands



Creating statistics



Monitoring automated processes



The rest of this talk

My approach to developing packages

Why I think R has great features for

- ▶ Standardization and Reuse
- ▶ Rule-based data processing
- ▶ Modularity
- ▶ Building User Interfaces

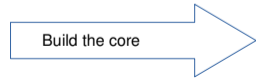


General Approach



Drill down to a fundamental (math) problem

```
R = {};  
for a in A do  
  i = 0;  
  d = f*({a});  
  while i < n ∧ ¬β(d) do  
    i = i + 1;  
    d = (f* ∘ Fi* ∘ Fi)(a);  
  end  
  if i < n ∨ β(d) then  
    R = R ∪ {(a, i, φ(d))};  
  end  
end
```

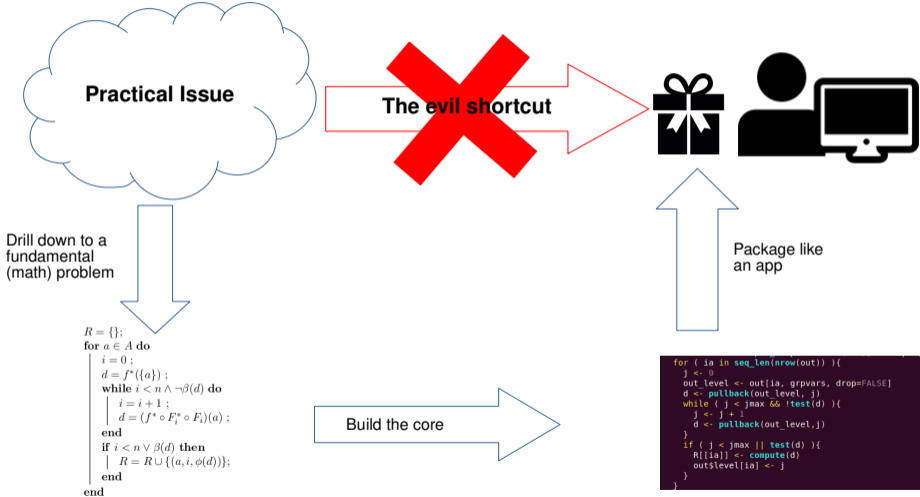


Package like an app

```
for ( ia in seq_len(nrow(out)) )(  
  j <- 0  
  out_level <- out[ia, grpvars, drop=FALSE]  
  d <- pullback(out_level, j)  
  while ( j < jmax && !test(d) )(  
    j <- j + 1  
    d <- pullback(out_level, j)  
  )  
  if ( j < jmax || test(d) )(  
    R[[ia]] <- compute(d)  
    out$level[ia] <- j  
  )  
)
```



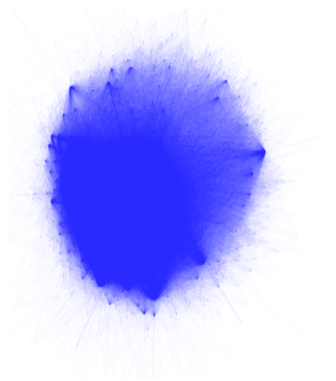
Approach



Standardization and Reuse



Standardization and Reuse with R



CRAN dependency network
(giant component)

- ▶ 19k packages
- ▶ 108k hard dependencies
- ▶ Highly standardized
 - ▶ Description and organization
 - ▶ Dependencies
 - ▶ Documentation
- ▶ Each package passes checks:
 - ▶ tests, examples
 - ▶ documentation, urls
 - ▶ code sanity
- ▶ Works on Windows, Linux, MacOS
- ▶ Works for previous, current, and development release of R
- ▶ **All dependencies resolve**
- ▶ **4 volunteers** + payed students



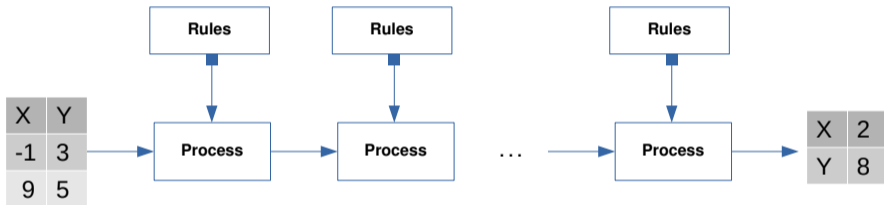


Rule-based data processing

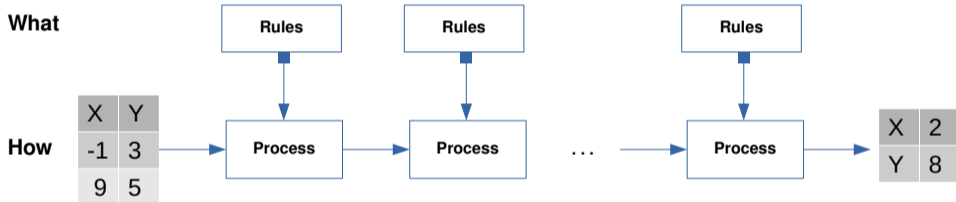
Rule-based data processing

What

How

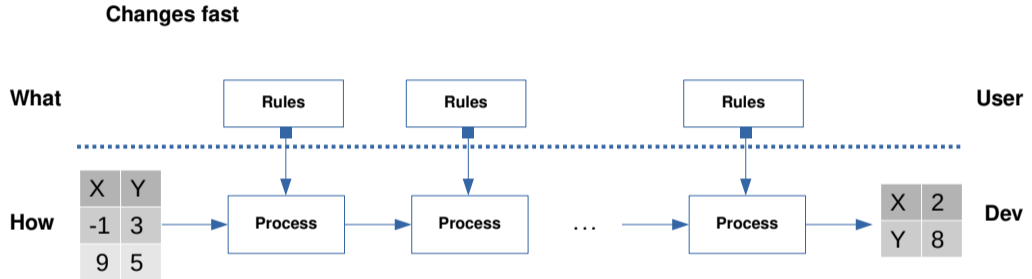


Rule-based data processing



```
data |>  
  dplyr::filter(x > 0) |>  
  dplyr::summarise(X = mean(X), Y=mean(Y)) |>  
  t()
```

Rule-based data processing



Changes slow

Rules should be documented; version controlled; stored and maintained externally.

The validate package



DEMO



The validate package

Approach

1. Theory: $v : D^K \hookrightarrow \{T, F\}$
2. Implementation of core theory into R
3. Design user interface / workflow
 - ▶ CRUD operations
 - ▶ Selecting
 - ▶ Investigating (e.g. `variables()`)
 - ▶ Validation + summarization of results



The validate package

Approach

1. Theory: $v : D^K \hookrightarrow \{T, F\}$
2. Implementation of core theory into R
3. Design user interface / workflow
 - ▶ CRUD operations
 - ▶ Selecting
 - ▶ Investigating (e.g. `variables()`)
 - ▶ Validation + summarization of results

Result

1. User-defined rules.
2. Easy to learn/get started.
3. Backend extensible (e.g. `validatedb`)
4. Use rules as input for other packages. (e.g. `errorlocate`)
5. Reason, investigate, manipulate rules and rulesets.



The validate package

Approach

1. Theory: $v : D^K \hookrightarrow \{T, F\}$
2. Implementation of core theory into R
3. Design user interface / workflow
 - ▶ CRUD operations
 - ▶ Selecting
 - ▶ Investigating (e.g. `variables()`)
 - ▶ Validation + summarization of results

NB

In a 2020 we demonstrated that all 'generic' data validation rules for the ESS be expressed with `validate`. <https://github.com/SNStatComp/GenericValidationRules>

Result

1. User-defined rules.
2. Easy to learn/get started.
3. Backend extensible (e.g. `validatedb`)
4. Use rules as input for other packages. (e.g. `errorlocate`)
5. Reason, investigate, manipulate rules and rulesets.



Contrasting approach (e.g. PointBlank, DataMaid)

Approach

1. Collect individual validation cases.
2. Implement each case in a function.



Contrasting approach (e.g. PointBlank, DataMaid)



Approach

1. Collect individual validation cases.
2. Implement each case in a function.

Result

1. No user-defined rules
2. Need to read manual for each function
3. Rules not reusable accros packages
4. No (systematic) manipulation or investigation of rules and rulesets.

Why R?

R has the 'reflexive' property

```
head(cars, 3)
```

	speed	dist
1	4	2
2	4	10
3	7	4

```
r <- expression(dist > speed)
eval(r, head(cars,3))
```

```
[1] FALSE TRUE FALSE
```

```
r[[1]][[1]] == ">"
```

```
[1] TRUE
```



Why R?

R has the 'reflexive' property

```
head(cars, 3)
```

```
  speed dist
1     4    2
2     4   10
3     7    4
```

```
r <- expression(dist > speed)
eval(r, head(cars,3))
```

```
[1] FALSE TRUE FALSE
```

```
r[[1]][[1]] == ">"
```

```
[1] TRUE
```

Anatomy of validate (pseudocode)

```
# parse
rules <- parse(file="rulefile.R")
v <- new_validator()
for ( r in rules ){
  if (is_validating(r)){
    v <- v + r
  }
}

# eval
out <- new_validation()
for (r in v){
  out <- out + eval(r, data)
}
```



More information

Wiley StatsRef:
Statistics Reference Online



Data Validation

Mark P.J. van der Loo and Edwin de Jonge

Keywords: data quality, data cleaning

Abstract. Data validation is the activity where one decides whether or not a particular data set is fit for a given purpose. Formulating the requirements that drive this decision process allows for automated communication of the requirements, automation of the decision process, and explicit ways to monitor and manage the decision process itself. The purpose of this article is to formalize the definition of data validation and to demonstrate some of the properties that can be derived from this definition. In particular, it is shown how a formal view of the concept permits a classification of data quality requirements, allowing them to be ordered in increasing levels of complexity. Some substitutes arising from combining possibly empty such requirements are pointed out as well.

Informally, data validation is the activity where one decides whether or not a particular data set is fit for a given purpose. The decision is based on testing observed data against given requirements that a plausible data set is assumed to satisfy. Examples of given requirements range widely. They include natural limits on variables (weight cannot be negative), restrictions on combinations of multiple variables (a man cannot be pregnant), combinations of multiple variables (a mother cannot be younger than her child), and combinations of multiple data sources (largest value of country A from country B must equal the largest value of country B from country A). Besides the natural logical constraints mentioned in the examples, there are other rules or constraints based on human experience. For example, one may not expect a certain measure (such as income) to be 10% or greater. Thus, the 10% limit does not represent a physical impossibility but rather a check based on past experience. One can never decide in the end whether a data set is usable for its intended purpose, or not, such as communication or legal testing.

The purpose of this article is to formalize the definition of data validation and to demonstrate some of the properties that can be derived from this definition. In particular, it is shown how a formal view of the concept permits a classification of data validation rules (constraints), allowing them to be ordered in increasing levels of "complexity." Here, the term "complexity" refers to the amount of different types of information necessary to make a validation rule. A formal definition also permits development of tools for automated validation and automated reasoning about data validation^{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100}.

¹Mark P.J. van der Loo, Edwin de Jonge, *Journal of Statistical Software*, 2021, 97:1-22.
²Mark P.J. van der Loo, Edwin de Jonge, *Journal of Statistical Software*, 2021, 97:1-22.
³Mark P.J. van der Loo, Edwin de Jonge, *Journal of Statistical Software*, 2021, 97:1-22.



Journal of Statistical Software

March 2021, Volume 97, Issue 01 doi:10.18637/jss.v097.i01

Data Validation Infrastructure for R

Mark P. J. van der Loo
Statistiek Netherlands

Edwin de Jonge
Statistiek Netherlands

Abstract

Checking data quality against domain knowledge is a common activity that pervades statistical analysis from raw data to output. The R package `validate` facilitates this task by capturing and applying expert knowledge in the form of validation rules. Logical restrictions on variables, records, or data sets that should be satisfied before they are considered valid input for further analysis. In the `validate` package, validation rules are objects of computation that can be manipulated, investigated, and validated with data or segments of a data set. The results of a validation are then available for further investigation, communication or visualization. Validation rules can also be combined with variables and demonstrated and they may be shared or retrieved from external sources, such as text files or database formats. This data validation infrastructure allows for systematic, semi-formal definition of data quality requirements that can be tested for various segments of a data set or by data correction algorithms that are parameterized by validation rules.

Keywords: data checking, data quality, data cleaning, R.

1. Introduction

Checking whether data satisfy assumptions based on domain knowledge pervades data analysis. Whether it is raw data, cleaned up data, or output of a statistical calculation, data analysts have to scrutinize data sets at every stage to ensure that they can be used for reporting or further computation. We refer to this procedure of investigating the quality of a data set and deciding whether it is fit for purpose as a "data validation" procedure. Many things can go wrong while creating, gathering, or processing data. Accordingly there are many types of checks that can be performed. One usually distinguishes between technical checks that are related to data structure and data type, and checks that are related to the topics described by the data. Examples of technical checks include testing whether an "age" variable is numeric, whether all necessary variables are present, or whether the identifying



modernstatIS

UNITED NATIONS ECONOMIC COMMISSION FOR EUROPE
CONFERENCE OF EUROPEAN STATISTICIANS
Expert meeting on Statistical Data Editing
3-7 October 2021, (virtual)

Rule Management

Mark van der Loo, Edwin de Jonge, Olof ten Bosch (Statistiek Netherlands, The Netherlands)
mjv@statistiek.nl

1. INTRODUCTION

1. An important aspect in the modernization of statistical production systems is the idea that subject matter knowledge should be separated as much as possible from technical (IT) knowledge. This is apparent in the Common Statistical Production Architecture (ModernStatIS, 2021) which promotes a separation between the description of business functions and implementation detail. It also promotes a modular architecture where building a production system in principle amounts to assembling a set of prebuilt modules for different business functions and parameterizing them with business logic for the systems at hand.
2. In practice these considerations lead to rule-based data processing systems. In the area of data editing this is outlined more at least since the seminal paper of Felling and Hain (1975) on semi-automation, statistical offices have worked with rule sets to engage subject matter knowledge. The idea of rule-based processing is also applied in the area of conditional, domain-specific correction rules (Poulsen et al. 1993) to make domain knowledge explicit and to make production processes reproducible and transparent. Other recent developments include the development of the Validation and Transformation Language (VTL, 2021) and the development of data validation and rule-based data cleaning engines based in R and Python (van der Loo and de Jonge, 2021a, 2021b, 2021c).
3. The advent of rule-based production systems, as well as the ongoing integration of production systems across institutes naturally leads to the issue of rule management. A need arises to manage and maintain rule sets that drive production processes. In these projects, rule management systems offer great advantages to statistical agencies. They facilitate reusability of production, exchange and reuse of formalized domain knowledge, and offer the possibility of reusing rule sets across production systems.
4. The idea of rule management systems is currently developed in several contexts. For example, Eurostat is building a repository for the Exchange of data validation rules within the European Statistical System. Within Statistiek Netherlands, two programmes for managing social and economic statistics aim to create modular and rule-based production systems, which calls for rule management systems.
5. This paper contributes to the current discussion by starting with the definition of a few new terms, that express what a user might expect from a repository. Next, we try to formalize the discussion by formalizing the concept of a rules and rule sets. We find that the central object of interest in the concept of a rule response, and related list of rules. Next, we address a basic set



MPJ van der Loo, E de Jonge (2020). Data Validation. In Wiley StatsRef: Statistics Reference Online, pages 1-7. American Cancer Society.

MPJ van der Loo, E de Jonge (2021). Data Validation Infrastructure for R. Journal of Statistical Software 1-22 97

M.P.J. van der Loo, E. de Jonge, K.O. ten Bosch (2022). Rule Management. UNECE Expert Meeting on Statistical Data Editing.



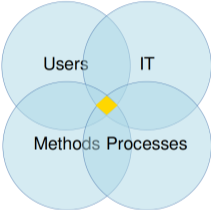


Modularity



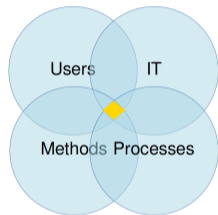
Why Modularity is Difficult

1. Perspectives on Modules

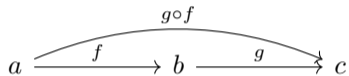


Why Modularity is Difficult

1. Perspectives on Modules

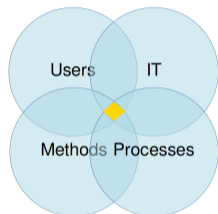


2. Modularity \neq Composability



Why Modularity is Difficult

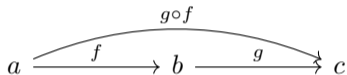
1. Perspectives on Modules



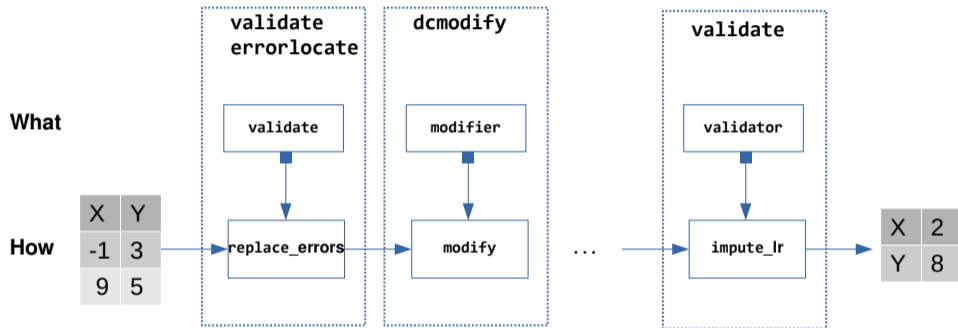
Ideal Modules

Are easy to understand and use (app-like); composable; implement a single piece of methodology; are controllable from the outside (rules/parameters).

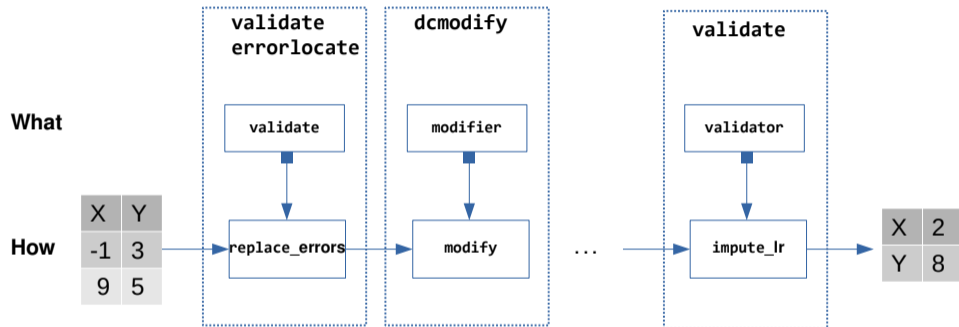
2. Modularity \neq Composability



Examples of modules for data processing



Examples of modules for data processing



Note

- ▶ Composing from left to right
- ▶ Modules can be independently added or removed



A module for process monitoring (logging)?



What

Rules

Rules

Rules

How

X	Y
-1	3
9	5

Process

Process

...

Process

X	2
Y	8

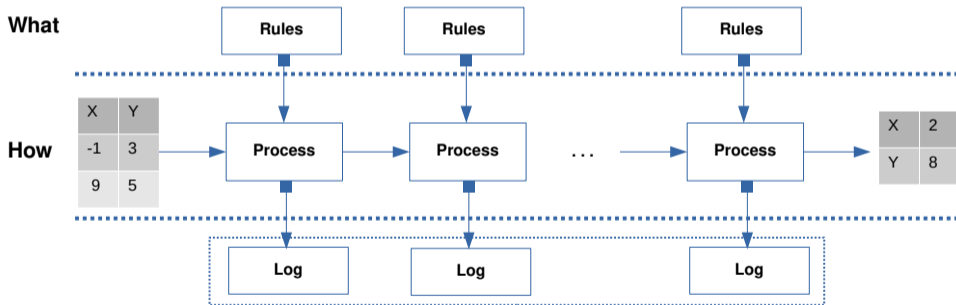
Log

Log

Log



A module for process monitoring (logging)?



Question

- ▶ How to implement a module that can be independently added for logging?



It can be done: `lumberjack`



DEMO

How does that work? 1: Adding a second data stream



Concept

A sequence of R expressions e_i ; $i = 1, 2, \dots, n$, executed in order can be considered as one long composed expression

$$e_n \circ e_{n-1} \circ \dots \circ e_1.$$

The idea is to replace this boring composition operator with an interesting one.



How does that work? 1: Adding a second data stream

source() (schematically)

```
expressions <- parse("myfile.R")
for (e in expressions){
  eval(e)
}
```



How does that work? 1: Adding a second data stream

source() (schematically)

```
expressions <- parse("myfile.R")
for (e in expressions){
  eval(e)
}
```

run_file() (schematically)

```
expressions <- parse("myfile.R")
n <- 0
for (e in expressions){
  eval(e)
  n <- n + 1
}
printf("Counted %d expressions\n",n)
```



How does that work? 2: communication from user stream to log stream

```
# myfile.R
x <- runif(1)
start_counting()
y <- 20
z <- x + y
```



How does that work? 2: communication from user stream to log stream

```
# myfile.R
x <- runif(1)
start_counting()
y <- 20
z <- x + y
```

Concept

A special expression (like `start_counting()`) changes the composition operator.

$$e_n \circ e_{n-1} \circ \dots \circ e_k \circ \dots \circ e_1.$$



How does that work? 2: communication from user stream to log stream

```
# myfile.R
x <- runif(1)
start_counting()
y <- 20
z <- x + y
```

Concept

A special expression (like `start_counting()`) changes the composition operator.

$$e_n \circ e_{n-1} \circ \dots \circ e_k \circ \dots \circ e_1.$$

- ▶ ~~Analyze Abstract Syntax Tree?~~ `if(x >= 0.5) start_counting()`
- ▶ Smuggle information from the special expression to the file runner.



Creating a smuggler with higher order functions

```
make_smuggler <- function(fun, env){  
  function(...){  
    env$result <- fun(...) # smuggle the result of 'fun' to 'env'  
    env$result  
  }  
}
```



Creating a smuggler with higher order functions

```
make_smuggler <- function(fun, env){  
  function(...){  
    env$result <- fun(...) # smuggle the result of 'fun' to 'env'  
    env$result  
  }  
}
```

```
store <- new.env()  
mymean <- make_smuggler(mean, store)  
  
mymean(1:3) # works just like 'mean'
```

```
[1] 2
```

```
store$result # but the result is also copied into 'store'
```

```
[1] 2
```

markvanderloo.eu



Using a smuggler in run_file()

```
start_counting <- function() return(TRUE)
```

```
run_file <- function(rfile){  
  runtime <- new.env()  
  store    <- new.env()  
  
  runtime$start_counting <- smuggler(start_counting, store)  
  
  expressions <- parse(rfile)  
  n <- 0  
  for (e in expressions){  
    eval(e, runtime)  
    if (isTRUE(store$result)) n <- n + 1  
  }  
  printf("Executed %d expressions", n)  
}
```



Design of lumberjack

Approach

1. Theory:
 - ▶ custom file runner
 - ▶ local side effect (smuggler)
 - ▶ local masking
2. Implement core into R
3. Design user interface
 - ▶ start/stop/pause/dump
 - ▶ Logger objects and interface



Design of lumberjack



Approach

1. Theory:
 - ▶ custom file runner
 - ▶ local side effect (smuggler)
 - ▶ local masking
2. Implement core into R
3. Design user interface
 - ▶ start/stop/pause/dump
 - ▶ Logger objects and interface

Result

1. Easy to get started/learn
2. Extensible with new loggers
 - ▶ Integration with validate
3. Composable (1 LoC)
4. Clean separation between runtime environment and logging environment.

Contrasting approaches: most logging packages

Approach

1. Define specific logging use cases
 - ▶ fixed output format/type
 - ▶ fixed output target/type
 - ▶ logging level
2. Implement as R functions



Contrasting approaches: most logging packages



Approach

1. Define specific logging use cases
 - ▶ fixed output format/type
 - ▶ fixed output target/type
 - ▶ logging level
2. Implement as R functions

Result

1. Easy to get started/learn
2. Need to insert code in several places
3. Usually not extensible w/new loggers
4. Logging takes place in runtime environment

NB:

5. Same approach used in `tinytest`

Modularity: Why R?

All modules

- ▶ R package system

'cross-cutting' modules, like logging

- ▶ Higher-order functions (func. lang.)
- ▶ Reflexive: parse/eval and masking
- ▶ Environment as smugglers' path



More information

18.S097: Programming with Categories

IAP 2020



B. Fong, B Milewski, D. Spivak (2020)
Programming with categories; online lectures
<http://brendanfong.com/programmingcats.html>



B. Milewski (2019) Category theory for programmers. Blurb.com.

markvanderloo.eu

COPYRIGHTED RESEARCH ARTICLE

42

A Method for Deriving Information from Running R Code

by Mark P.J. van der Loo

Abstract It is often useful to tap information from a running R script. Common use cases include monitoring the consumption of resources (time, memory) and logging. Perhaps less obvious cases include tracking changes in the collection and processing of data. In this paper, we demonstrate an approach that abstracts the collection and processing of such secondary information from the running R script. Our approach is based on a combination of three elements. The first element is to build a convenient way to evaluate code. The second is labeled local monitoring and is achieved temporarily marking a user-facing function as an alternative version of it called. The third element we shall deal with later. This refers to the fact that the marking function requires information on the secondary information flow without altering a global state. The result is a method for building systems in pure R that lets users create and control secondary flows of information with minimal impact on their workflow and no global side effects.

Introduction

The R language provides a convenient language to read, manipulate, and write data in the form of scripts. As with any other scripted language, an R script gives a description of data manipulation activities, one after the other, when read from top to bottom. Alternatively, we can think of an R script as a one-dimensional (manifestation of) data flowing from one processing step to the next, where intermediate variables or pipe operators carry data from one treatment to the next.

We run into limitations of this one-dimensional view when we want to produce data flows that are somehow 'orthogonal' to the flow of the data being treated. For example, we may wish to follow the state of a variable while a script is being executed, upon or between (logging), or keep track of resource consumption. Indeed, the sequential (one-dimensionally) nature of a script forces us to introduce extra operations between the data processing code.

As an example, consider a code fragment where the variable `x` is manipulated.

```
x[] > threshold() <- threshold()
x[] > threshold() <- median(x, na.rm=TRUE)
```

In the first statement, every value above a certain threshold is replaced with a fixed value, and every missing value are replaced with the median of the completed cases. It is interesting to know how an appropriate system, say the amount of `x`, evolves as it gets processed. The indicator to use to do this is to add the code by adding statements to the script that collect the desired information.

```
mean <- mean(x, na.rm=TRUE)
x[] > threshold() <- threshold()
mean <- c(mean, mean(x, na.rm=TRUE))
x[] > threshold() <- median(x, na.rm=TRUE)
mean <- c(mean, mean(x, na.rm=TRUE))
```

This solution changes the script by inserting expressions that are not necessary for its main purpose. Moreover, the tracking statements are repetitive, which violates some form of abstraction.

A more general picture of what we would like to achieve is given in Figure 1. The 'primary data flow' is denoted by `x` as in the script. In the previous example, this means parameter `W`. When the script runs, some kind of logging information, which we label the 'secondary data flow' is derived implicitly by an abstraction layer.

Creating an abstraction layer means that concerns between primary and secondary data flows are separated as much as possible. In particular, we want to prevent the abstraction layer from inspecting or altering the user code that describes the primary data flow. Furthermore, we would like the user to have some control over the secondary data flow within the script, for example, to start, stop, or parameterize the monitoring. This should be done with care, however, as it should not only be global side effects. This means that neither the user nor the abstraction layer for the secondary data flow should have to manipulate or read global variables, options, or other site-environmental settings to convey information from one flow to the other. Ideally, we would like the availability of a secondary data flow as a normal situation. This means we wish to avoid using signaling conditions (e.g., warnings or errors) to convey information between the flows unless there is an actual exceptional condition such as an error.

The R Journal Vol. 13/1, June 2021

ISSN 2073-4859

Mark P.J. van der Loo (2021). A Method for Deriving Information from Running R Code. The R Journal 13 42–52

Modernization of statistical production with R



Journal of Statistical Software

May 2021, Volume 98, Issue 1

doi:10.18637/jss.v098.i01



Monitoring Data in R with the lumberjack Package

Mark P. J. van der Loo
Statistiek, Netherlands

Abstract

Monitoring data while it is processed and transformed can yield detailed insight into the dynamics of a (running) production system. The lumberjack package is a lightweight R package allowing users to follow how an R object is transformed as it is manipulated by R code. The package abstracts all logging code from the user, who only needs to specify which objects are logged and what information should be logged. A few default loggers are included with the package but the package is extensible through user-defined logger objects.

Keywords: data quality, process monitoring, logging, debugging, R.

1. Introduction

It is common practice to monitor a data analysis process while it is running, especially in production environments where analyses are run repeatedly on different but structurally comparable data sets. Following a running procedure is usually done with some form of logging system, where the running process updates a log that can be tracked by users as it proceeds.

One can distinguish two types of monitoring. On the one hand there is process logging, or just logging for short. Here, the running system notifies users of progress and significant events, usually by writing short time-stamped messages to a file (perhaps "file" can be a flat text file, database, screen or any other device accepting such input). The idea of these messages is to signal whether procedures have concluded successfully, and if they have not, to report what went wrong. Such information is highly valuable in post-mortem investigations for example when a production script has crashed. On the other hand there is tracing where the state of variables is followed over the course of the process. Tracing is usually applied at the development stage or a debugging tool, often using an interactive interface tool to run the code line by line while inspecting the state of variables. One of the purposes of this paper is to demonstrate that targeted forms of automated tracing can be useful at the production stage as well.

MPJ van der Loo (2021). Monitoring data in R with the lumberjack package. Journal of Statistical Software 98 1–11

uRos2023 Bucharest





User interface

Example: simputation

Example data

	staff	turnover	other.rev	total.rev	staff.costs
1	75	NA	NA	1130	NA
2	9	1607	NA	1607	131
3	NA	6886	-33	6919	324



Example: simputation



Example data

	staff	turnover	other.rev	total.rev	staff.costs
1	75	NA	NA	1130	NA
2	9	1607	NA	1607	131
3	NA	6886	-33	6919	324

Issues

- ▶ Need for fall-through scenario for imputation
- ▶ Multiple variables to impute
- ▶ Multiple models

Imputation in R

Specialized packages

- ▶ 100+ available (VIM, mice, Amelia, mi, ...)
- ▶ Interfaces vary (a lot)

DIY with model/predict

```
m          <- lm(Y ~ X, data = mydata)
ina        <- is.na(mydata$Y)
mydata[ina, "Y"] <- predict(m, newdata = mydata[ina,])
# YMMV, be cause how 'predict' works, depends on the model
```

Result

- ▶ Lots of 'boilerplate' code needed, covering many cases
- ▶ Hard to experiment and test



Idea of the simputation package

Provide

- ▶ a *uniform interface*,
- ▶ with *consistent behaviour*,
- ▶ across *commonly used methodologies*

To facilitate

- ▶ experimentation
- ▶ configuration for production



The imputation interface

```
impute_<model>(data  
  , <imputed vars> ~ <predictor vars>  
  , [options])
```



The imputation interface

```
impute_<model>(data  
  , <imputed vars> ~ <predictor vars>  
  , [options])
```

Example: linear model imputation

```
impute_lm(ret, other.rev ~ turnover) |> head(3)
```

	staff	turnover	other.rev	total.rev	staff.costs
1	75	NA	NA	1130	NA
2	9	1607	5427.113	1607	131
3	NA	6886	-33.000	6919	324



Fall-through scenario by chaining imputations



```
ret |>  
  impute_lm(other.rev ~ turnover + staff) |>  
  impute_lm(other.rev ~ staff) |>  
  head(3)
```

	staff	turnover	other.rev	total.rev	staff.costs
1	75	NA	13914.261	1130	NA
2	9	1607	6089.698	1607	131
3	NA	6886	-33.000	6919	324

Other features

- ▶ Impute multiple variables with the same model
- ▶ Groupwise imputation
- ▶ Multivariate methods (e.g. missForest, EM)



Approach



Approach

1. Essentials of specification
2. Map to R features (formula-data)
3. Build on top of existing packages



Approach



Approach

1. Essentials of specification
2. Map to R features (formula-data)
3. Build on top of existing packages

Result

1. Easy to learn and use :-)
2. Fall-through scenarios supported :-)
3. Not driven by external rules :-/
4. Hard to extend :-)

Why R?

- ▶ Availability of many existing imputation methods
- ▶ Formula-data interface for model specification
- ▶ Packaging model, and CRAN





Summary



In my experience



Peeling off a practical issue until a math problem remains

- ▶ Is essential to achieve true modularity and composability;
- ▶ Truly separates concerns between user needs (domain knowledge) and programming.
- ▶ Yields extensible solutions;
- ▶ Creates interfaces that are almost automatically user-friendly



What makes R specifically suited for modernization?

High level of abstraction

Higher order functions, reflexivity facilitate true **modularity** and **composability**.



What makes R specifically suited for modernization?

High level of abstraction

Higher order functions, reflexivity facilitate true **modularity** and **composability**.

Functional language and reflexivity

Allows for quick development of **domain-specific languages** embedded into R



What makes R specifically suited for modernization?

High level of abstraction

Higher order functions, reflexivity facilitate true **modularity** and **composability**.

Functional language and reflexivity

Allows for quick development of **domain-specific languages** embedded into R

Abstractions for model-building

Formula-data interface facilitate **user-friendly interfaces**.



What makes R specifically suited for modernization?

High level of abstraction

Higher order functions, reflexivity facilitate true **modularity** and **composability**.

Functional language and reflexivity

Allows for quick development of **domain-specific languages** embedded into R

Abstractions for model-building

Formula-data interface facilitate **user-friendly interfaces**.

Packaging system and CRAN

Nothing beats CRAN in terms of consistency across dependencies, quality checks and **standardization**.



What makes R specifically suited for modernization?

High level of abstraction

Higher order functions, reflexivity facilitate true **modularity** and **composability**.

Functional language and reflexivity

Allows for quick development of **domain-specific languages** embedded into R

Abstractions for model-building

Formula-data interface facilitate **user-friendly interfaces**.

Packaging system and CRAN

Nothing beats CRAN in terms of consistency across dependencies, quality checks and **standardization**.

The R community (you!)



Thank you



markvanderloo.eu/publications.html

