

Galaaz Manual

Coupling Ruby (JRuby) and GNU R for data science

Rodrigo Botafogo

2026

Contents

1	Introduction	3
1.1	What does Galaaz mean	4
2	Command-line tools (bin/)	5
2.1	Interactive use, examples, and Rake	5
2.2	gKnit and document drafts	6
2.3	Tests	6
2.4	Other	6
3	System Compatibility	7
4	Dependencies	7
5	Installation	7
5.1	Windows + WSL2 (optional: Docker / R in a container)	8
6	Usage	8
7	JRuby, multithreading, and the R bridge	9
7.1	Long-running R calls and a completion block	9
8	Accessing R from Ruby	10
9	gKnitting a Document	13
9.1	gKnit and R markdown	15
9.2	The Yaml header	15
9.3	Choosing the output format when calling gknit	15
9.4	R Markdown formatting	16
9.5	Headers	16

9.6	Lists	16
9.7	R chunks	17
9.8	R Graphics with ggplot	18
9.9	Ruby chunks	20
9.10	Inline Ruby code	20
9.10.1	The ‘outputs’ function	21
9.10.2	HTML Output from Ruby Chunks	21
9.11	Including Ruby files in a chunk	23
9.12	Documenting Gems	25
9.13	Converting to PDF	27
9.14	Template based documents generation	27
10	Accessing R variables	28
11	Basic Data Types	29
11.1	Vector	29
11.1.1	Combining Vectors	31
11.1.2	Vector Arithmetic	31
11.1.3	Vector Indexing	32
11.1.4	Extracting Native Ruby Types from a Vector	33
11.2	Matrix	33
11.2.1	Indexing a Matrix	34
11.3	List	35
11.3.1	List Indexing	36
11.4	Data Frame	36
11.4.1	Data Frame Indexing	37
12	Writing Expressions in Galaaz	38
12.1	Expressions from operators	39
12.2	Expressions with R methods	40
12.3	Evaluating an Expression	40
13	Manipulating Data	41
13.1	Filtering rows with Filter	41
13.2	Logical Operators	42
13.3	Filtering with NA (Not Available)	43
13.4	Arrange Rows with arrange	43
13.5	Selecting columns	44

13.6 Add variables to a dataframe with ‘mutate’	46
13.7 Summarising data	46
14 Using Data Table	47
15 Apache Arrow	50
15.1 Other R::Arrow helpers	50
15.2 Example: many Ruby rows → Arrow in R → grouped statistics	51
16 Bioconductor and DESeq2	52
16.1 Installing Bioconductor packages	52
16.2 Example: DESeq2 on the airway dataset	52
17 Performance	54
18 Graphics in Galaaz	55
19 Coding with Tidyverse	57
19.1 Writing a function that applies to different data sets	59
19.2 Different expressions	60
19.3 Different input variables	62
19.4 Different input and output variable	63
19.5 Capturing multiple variables	64
19.6 Why does R require NSE and Galaaz does not?	64
19.7 Advanced dplyr features	65
20 Contributing	67
References	67

1 Introduction

Galaaz is a system for tightly coupling Ruby and R. Ruby is a powerful language, with a large community, a very large set of libraries and great for web development. However, it lacks libraries for data science, statistics, scientific plotting and machine learning. On the other hand, R is considered one of the most powerful languages for solving all of the above problems. **Python** is a strong competitor: NumPy, pandas, SciPy, and scikit-learn are widely used building blocks, and **PyPI** hosts many thousands of other packages for numerical work, machine learning, and beyond.

With Galaaz we do not intend to re-implement any of the scientific libraries in R, we allow for very tight coupling between the two languages to the point that the Ruby developer does not need to know that there is an R engine running.

According to Wikipedia “Ruby is a dynamic, interpreted, reflective, object-oriented, general-purpose programming language. It was designed and developed in the mid-1990s by Yukihiro”Matz” Matsumoto in Japan.” It reached high popularity with the development of Ruby on Rails (RoR) by David Heinemeier Hansson. RoR is a web application framework first released around 2005. It makes extensive use of Ruby’s metaprogramming features. With RoR, Ruby became very popular. According to Ruby’s place in the TIOBE index it peaked in popularity around 2008, then declined until 2015 when it started picking up again. Ruby remains a significant language in web development and general-purpose scripting.

Python, a language similar to Ruby, ranks 4th in the index. Java, C and C++ take the first three positions. Ruby is often criticized for its focus on web applications. But Ruby can do much more than just web applications. Yet, for scientific computing, Ruby lags behind Python and R. Python offers Django and similar frameworks for the web, plus NumPy, pandas, and a deep catalog of science and ML libraries. R is a free software environment for statistical computing and graphics with thousands of libraries for data analysis.

Until recently, there was no real perspective for Ruby to bridge this gap. Implementing a complete scientific computing infrastructure would take too long.

Galaaz 2.0 couples **JRuby** (Ruby on the JVM) with **GNU R**—the same R you use for CRAN and Bioconductor. Ruby and R run in **separate processes**; the **Galaaz bridge** sends requests to R and returns results to Ruby. From your point of view you still write Ruby: `R.c(...)`, `R.library('ggplot2')`, `~:mtcars`, and dplyr-style chains on R objects. You do not need to learn R syntax to get a lot done, though reading R documentation for individual packages remains useful.

Earlier experiments with Galaaz used Oracle’s **GraalVM** with TruffleRuby and FastR so that Ruby and R could share one runtime. That path is no longer the focus: **standard GNU R** gives full compatibility with the R package ecosystem (including compiled extensions and Bioconductor) while JRuby gives a mature Ruby with **real multithreading** for application and I/O code.

The bridge handles **communication and typing** between the two worlds; large tables can also flow through **Apache Arrow** on the R side when you use the optional helpers described later in this manual.

Library wrapping is a common way to bring features from one language into another. To improve performance, Python often wraps more efficient C libraries. For the Python developer, the existence of such C libraries is hidden. The problem with library wrapping is that for any new library, there is the need to handcraft a new wrapper.

Galaaz, instead of wrapping a single C or R library, wraps the whole R language in Ruby. Doing so, all thousands of R libraries are available immediately to Ruby developers without any new wrapping effort.

1.1 What does Galaaz mean

Galaaz is the Portuguese name for “Galahad”. From Wikipedia:

Sir Galahad (sometimes referred to as Galeas or Galath), in Arthurian legend, is a knight of King Arthur’s Round Table and one of the three achievers of the Holy Grail. He is the illegitimate son of Sir Lancelot and Elaine of Corbenic, and is renowned for his gallantry and purity as the most perfect of all knights. Emerging quite late in the medieval Arthurian tradition, Sir Galahad first appears in the

Lancelot-Grail cycle, and his story is taken up in later works such as the Post-Vulgate Cycle and Sir Thomas Malory's *Le Morte d'Arthur*. His name should not be mistaken with Galehaut, a different knight from Arthurian legend.

2 Command-line tools (bin/)

The Galaaz repository ships many helpers under **bin/**. When working from a **clone**, call them as **bin/<name>** from the project root (or **./bin/<name>**). If you install the **gem**, only a subset is guaranteed on your **PATH** (see the **gemspec**: **galaaz**, **gstudio**, **gknit**, **grun**, **gknit-draft**); for development and CI, prefer the **bin/** copies so JVM flags and paths stay correct.

Below, **current** (**Galaaz 2.0 + JRuby + GNU R**) means the tool is wired to **jruby** and **bin/galaaz_jruby_env.inc.sh** (or equivalent logic in Ruby via **lib/galaaz_jruby.rb**). **Legacy** means the script still targets **GraalVM** polyglot Ruby / FastR-era invocation and is **not** expected to work on a typical JRuby-only setup.

Table layout: names in the first column are **bin/** filenames (run as **bin/<name>** from the repo root). Long options and examples sit **outside** the tables so PDF columns stay readable.

```
## Warning in styling_latex_scale(out, table_info, "down"): Longtable cannot be
## resized.
```

Script	Role	2.0?
galaaz-bootstrap	WSL2 helper: Docker checks; optional TinyTeX or poppler for gKnit PDF.	Yes*
galaaz-jruby	JRuby with repo lib/ on LOAD_PATH and required JVM flags (e.g. Arrow).	Yes
galaaz_jruby_env.inc.sh	Sourced by bash wrappers; sets GALAAZ_REQUIRED_JRUBY_J_ARGS.	Yes†
install-tinytex	Install TinyTeX for PDF output.	Yes

* Where WSL/Docker apply. **galaaz-bootstrap flags:** **--check**, **--apply**, **--runtime** (docker | local | auto), **--[no-]prompt-doc-tools**.

† Not run directly.

galaaz-jruby examples (from repo root):

```
bin/galaaz-jruby my_script.rb
bin/galaaz-jruby -S rspec
```

2.1 Interactive use, examples, and Rake

```
## Warning in styling_latex_scale(out, table_info, "down"): Longtable cannot be
## resized.
```

Script	Role	2.0?
gstudio	IRB or Pry with Galaaz preloaded (JRuby + JVM flags).	Yes
run_example	Run one Ruby file using the same JRuby/JVM setup as tests.	Yes

galaaz	Forward arguments to rake (needs rake; usually JRuby).	Yes
--------	--	-----

2.2 gKnit and document drafts

```
## Warning in styling_latex_scale(out, table_info, "down"): Longtable cannot be
## resized.
```

Script	Role	2.0?
gknit	Knit .Rmd via JRuby and R Markdown render.	Yes
gknit-draft	Drafts from rticles-style templates; wrapper still uses legacy polyglot ruby.	Legacy
gknit-draft.rb	Ruby entry: GKnit.draft (use with JRuby + LOAD_PATH).	JRuby
gknit_Rscript	Polyglot Rscript launcher; hard-coded LOAD_PATH sample.	No

gknit CLI (see `gknit -h`): `--output_format`, `--output_file`, `--output_dir`, `--bridge_timeout_sec`, `--callback_timeout_ms`. If `--output_format` is omitted, the **first** YAML output: target wins.

Prefer **galaaz-jruby** for **gknit-draft** workflows until that wrapper matches the **gknit** stack.

2.3 Tests

```
## Warning in styling_latex_scale(out, table_info, "down"): Longtable cannot be
## resized.
```

Script	Role	2.0?
run_rspec	Top-level specs/*_spec.rb with spec_helper (see docs/testing.md).	Yes
run_all_rspec	Compile ext/new_bridge; run specs/ and new_bridge_specs/ together.	Yes
run_slow_rspec	Suites under slow-specs/ (read script header for spec_helper).	Yes
run_old_rspec	Legacy suites under old_specs/.	Yes
run_rspec_subset	Numbered subset 1–18 (Documentation/Spec_Subsets.md).	Yes

2.4 Other

```
## Warning in styling_latex_scale(out, table_info, "down"): Longtable cannot be
## resized.
```

Script	Role	2.0?
grun	Graal-era launcher: polyglot ruby with -jvm. Use galaaz-jruby -S instead.	No
gstudio_irb.rb / gstudio_pry.rb	Loaded by gstudio; not meant to be run standalone.	Yes

For day-to-day **2.0** use, rely on **bin/galaaz-jruby**, **bin/gstudio**, **bin/gknit**, **bin/run_example**, **bin/run_rspec** / **bin/run_all_rspec**, and **bin/galaaz-bootstrap** on WSL when using Dockerized R. Treat **grun**, **gknit_Rscript**, and the polyglot **ruby** invocation in **gknit-draft** as **legacy** until they are ported to the same JRuby path as **gknit**.

3 System Compatibility

Typical development and CI targets:

- **Linux** — recent Ubuntu LTS or comparable distributions (x86_64).
- **macOS** — recent releases with JRuby and GNU R available.
- **Windows** — use **WSL2** (same Linux stack as above); native Windows is not the primary target.

The native **gatekeeper** component under `ext/new_bridge` is built with `make` and a C++ toolchain; see the project **README** if compilation fails on your platform.

4 Dependencies

- **JRuby** — Ruby implementation on the JVM (Galaaz 2.0 is developed and tested with JRuby).
- A **Java runtime** (JDK) compatible with your JRuby version.
- **GNU R** — R on your PATH, with ability to install packages (CRAN / Bioconductor as needed).
- **Ruby gems** — from the Galaaz repository, `bundle install` (see the **Gemfile** / **gemspec**).
- Optional: **Docker** — if you run R in a container (common on WSL2); see bootstrap below.
- Optional R packages for examples in this manual — e.g. `ggplot2`, `dplyr`, `knitr`, `kableExtra`, `arrow`, Bioconductor tools such as **DESeq2** (installed the usual R way).

5 Installation

From a clone of the Galaaz repository:

1. Install **JRuby** and **GNU R** using your preferred package manager or Ruby version manager.
2. Install **bundler** if needed, then run `bundle install` in the repository root.
3. Build the bridge native code: `make -C ext/new_bridge all` (or `rake compile_gatekeeper`).
4. Run Ruby scripts with the project load path and JVM flags the project expects — the `bin/galaaz-jruby` wrapper sources `bin/galaaz-jruby_env.inc.sh` and adds `-I lib`. This matters especially for **Apache Arrow** integration (see `docs/testing.md`).
5. Ensure **R** starts GNU R and can install packages (network access to CRAN mirrors when you first call `R.install_and_loads`).

For **gKnit**, **knitr**, **rmarkdown**, and LaTeX (PDF output), install the corresponding R packages and a TeX distribution if you need PDF; the repository includes helpers such as `bin/install-tinytex` where appropriate.

A **table of all bin/ scripts** (bootstrap, JRuby wrapper, gstudio, gknit, test runners, and which ones are legacy) is in the section **Command-line tools (bin/)** earlier in this manual.

5.1 Windows + WSL2 (optional: Docker / R in a container)

If you run Galaaz on Windows through WSL2 and want containerized R instances, Docker Desktop is the supported setup.

1. Install Docker Desktop on Windows:
 - <https://www.docker.com/products/docker-desktop/>
2. Open Docker Desktop and enable WSL integration:
 - Settings > Resources > WSL Integration
 - Enable integration for your target distro
 - Apply & Restart Docker Desktop
3. In WSL, run Galaaz bootstrap:

```
ruby bin/galaaz-bootstrap --apply ruby bin/galaaz-bootstrap --check
```

Expected result: - docker CLI available - docker compose available - docker daemon reachable (docker info works)

If bootstrap reports daemon is unreachable, check Docker Desktop is running and WSL integration is enabled for the distro where Galaaz is installed.

6 Usage

- Interactive shell: use 'gstudio' on the command line

```
gstudio
```

```
vec = R.c(1, 2, 3, 4)
puts vec
```

```
## [1] 1 2 3 4
```

- Run all specs

```
galaaz specs:all
```

- Run graphics slideshow (80+ graphics)

```
galaaz sthda:all
```

- Run labs from Introduction to Statistical Learning with R

```
galaaz islr:all
```

- See all available examples

```
galaaz -T
```


Shows a list with all available executable tasks. To execute a task, substitute the ‘rake’ word in the list with ‘galaaz’. For instance, the following line shows up after ‘galaaz -T’

```
rake master_list:scatter_plot # scatter_plot from:...
```

execute

```
galaaz master_list:scatter_plot
```

7 JRuby, multithreading, and the R bridge

Galaaz 2.0 runs Ruby on **JRuby**, so your application can use **real parallel threads** for I/O-bound work (HTTP clients, database connections, message consumers, and so on). R itself is still executed in a **single GNU R process** behind the Galaaz bridge.

When several Ruby threads call into R at the same time, the bridge **serializes** those calls: each request is matched to a reply using an internal per-call **queue**, so you do not need to add your own mutex around every `R.foo` from application threads. (You should still use normal Ruby synchronization when **Ruby** data structures are shared between threads—for example, when appending rows from each thread into a shared array before sending them to R.)

A practical pattern is:

1. Use threads (or a connection pool) to read from **multiple databases or shards** in parallel.
2. Merge the rows in Ruby under a **Mutex** if you collect into one structure.
3. Hand the merged table to R **once** (for example with `R::Arrow.from_ruby_batches` and `dplyr`, or by building a data frame) so heavy statistics run in R with fewer bridge round-trips.

A runnable sketch lives in `examples/multithread_shards_to_r/shards_to_r.rb` (simulated shard queries; swap in your DB driver). For concurrency tests on the bridge itself, see `specs/bridge_concurrent_spec.rb` and `specs/arrow_from_ruby_batches_spec.rb`.

7.1 Long-running R calls and a completion block

For R work that can take a long time, the bridge can avoid a Ruby-side **wait timeout** by scheduling the call and resuming in a **block** when the RET arrives.

- `R.eval_r_async(code, timeout: nil) { |result| ... }` — string eval; on success, `result.value` is the same formatted string as `R.eval_r` (use `timeout: nil` for no Ruby-side limit).
- `R::Async.<cname>(...) { |result| ... }` — same dispatch as `R.<cname>(...)`, but async; on success, `result.value` is an `R::Object` (or unboxed Ruby value / Symbol), like synchronous `R.<cname>`. Optional keyword `timeout:` applies a Ruby-side wait limit (completion receives `NewBridge::SessionClient::TimeoutError` if R is too slow).

Important: `R.foo(...)` `{ |x| }` is already used for `dplyr`-style scopes (`R::Support.new_scope`), so async R calls must use `R::Async` or `R.eval_r_async`, not a bare `R.foo` with a **block**.

`NewBridge::EvalResult` exposes `#ok?`, `#value`, and `#error`. The completion block runs on a **background thread** (not the bridge reader thread).

The example below is **plain Ruby** (no Rails). The R snippet sleeps (standing in for heavy work) and then returns an integer so the success branch shows a **non-nil** value. (`Sys.sleep` alone returns **NULL** in R; on success `result.value` is then **nil** in Ruby—that is expected, not a bridge error.)

```
require 'thread'

completion = Queue.new

R.eval_r_async('{ Sys.sleep(0.3); 42L }', timeout: nil) do |result|
  if result.ok?
    puts "[completion] R finished; eval_r-style value: #{result.value.inspect}"
  else
    puts "[completion] R/bridge error: #{result.error.class}: #{result.error.message}"
  end
  completion.push(:done)
end

3.times do |i|
  puts "[main] other Ruby work step #{i + 1}"
  sleep 0.05
end

completion.pop
puts "[main] R completion has run; exiting."
```

```
## [main] other Ruby work step 1
## [main] other Ruby work step 2
## [main] other Ruby work step 3
## [completion] R finished; eval_r-style value: "[1] 42"
## [main] R completion has run; exiting.
```

In a **web application**, the HTTP response usually ends before R finishes, so you would not `Queue#pop` in the controller; you would persist an identifier, let the completion block write the outcome to storage, and notify the client (poll, WebSocket, Turbo Stream, etc.). The plain Ruby pattern above is only to show **when** the result exists (inside the block, or after data written there is observed elsewhere). Runnable specs live in `new_bridge_specs/eval_r_async_spec.rb`.

8 Accessing R from Ruby

One of the nice aspects of Galaaz is that variables and functions defined in R can be easily accessed from Ruby. For instance, to access the `mtcars` data frame from R in Ruby, we use the symbol `:mtcars` preceded by the `~` operator: `~:mtcars` retrieves the value of the `mtcars` object in R.

```
puts ~:mtcars
```

```
##               mpg cyl  disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4      21.0   6 160.0 110  3.90 2.620 16.46  0  1    4    4
```

## Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
## Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
## Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
## Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
## Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
## Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
## Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
## Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
## Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
## Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
## Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
## Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
## Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
## Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
## Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
## Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
## Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
## Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
## Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
## Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
## Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
## AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
## Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
## Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
## Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
## Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
## Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
## Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
## Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
## Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
## Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

To access an R function from Ruby, the R function needs to be preceded by `R.` scoping. Below we see an example of creating a `R::Vector` by calling the `'c'` R function

```
puts vec = R.c(1.0, 2.0, 3.0, 4.0)
```

```
## [1] 1 2 3 4
```

Note that `'vec'` is an object of type `R::Vector`:

```
puts vec.class
```

```
## R::Vector
```

Every object created by a call to an R function will be of a type that inherits from `R::Object`. In R, there is also a function `'class'`. In order to access that function we can call method `'rclass'` in the `R::Object`:

```
puts vec.rclass
```

```
## numeric
```

When working with `R::Object(s)`, it is possible to use the `'.'` operator to pipe operations. When using `'.'`, the object to which the `'.'` is applied becomes the first argument of the corresponding R function. For instance, function `'c'` in R, can be used to concatenate two vectors or more vectors (in R, there are no scalar values, scalars are converted to vectors of size 1. Within Galaaz, scalar parameter is converted to a size one vector):

```
puts R.c(vec, 10, 20, 30)
```

```
## [1]  1  2  3  4 10 20 30
```

The call above to the `'c'` function can also be done using `'.'` notation:

```
puts vec.c(10, 20, 30)
```

```
## [1]  1  2  3  4 10 20 30
```

We will talk about vector indexing in a later section. But notice here that indexing an `R::Vector` will return another `R::Vector`:

```
puts vec[1]
```

```
## [1] 1
```

Sometimes we want to index an `R::Object` and get back a Ruby object that is not wrapped in an `R::Object`, but the native Ruby object. For this, we can index the R object with the `'>'` operator:

```
puts vec >> 0
puts vec >> 2
```

```
## 1.0
```

```
## 3.0
```

It is also possible to call an R function with named arguments, by creating the function in Galaaz with named parameters. For instance, here is an example of creating a `'list'` with named elements:

```
puts R.list(first_name: "Rodrigo", last_name: "Botafogo")
```

```
## $first_name
```

```
## [1] "Rodrigo"
```

```
##
```

```
## $last_name
```

```
## [1] "Botafogo"
```

Many R functions receive another function as argument. For instance, method ‘map’ applies a function to every element of a vector. With Galaaz, it is possible to pass a Proc, Method or Lambda in place of the expected R function. In this next example, we will add 2 to every element of our previously created vector:

```
puts vec.map { |x| x + 2 }
```

```
## [1] 3 4 5 6
```

9 gKnitting a Document

This manual has been formatted using gKnit. gKnit uses knitr and R Markdown to knit a document in Ruby or R and output it in any of the available formats for R Markdown. gKnit runs with **JRuby**, **GNU R**, and Galaaz. In gKnit, Ruby variables are persisted between chunks, making it an ideal solution for literate programming. Also, since it is based on Galaaz, Ruby chunks can have access to R variables and combining Ruby with R in one document is natural.

The idea of “literate programming” was first introduced by Donald Knuth in the 1980’s (Knuth 1984). The main intention of this approach was to develop software interspersing macro snippets, traditional source code, and a natural language such as English in a document that could be compiled into executable code and at the same time easily read by a human developer. According to Knuth “The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style.”

The idea of literate programming evolved into the idea of reproducible research, in which all the data, software code, documentation, graphics etc. needed to reproduce the research and its reports could be included in a single document or set of documents that when distributed to peers could be rerun generating the same output and reports.

The R community has put a great deal of effort in reproducible research. In 2002, Sweave was introduced and it allowed mixing R code with LaTeX, generating high-quality PDF documents. A Sweave document could include code, the results of executing the code, graphics and text such that it contained the whole narrative to reproduce the research. In 2012, Knitr, developed by Yihui Xie from RStudio was released to replace Sweave and to consolidate in one single package the many extensions and add-on packages that were necessary for Sweave.

With Knitr, **R markdown** was also developed, an extension to the Markdown format. With **R markdown** and Knitr it is possible to generate reports in a multitude of formats such as HTML, Markdown, LaTeX, PDF, DVI, etc. **R markdown** also allows the use of multiple programming languages such as R, Ruby, Python, etc. in the same document.

In **R markdown**, text is interspersed with code chunks that can be executed and both the code and its results can become part of the final report. Although **R markdown** allows multiple programming languages in the same document, only R and Python (with the reticulate package) can persist variables between chunks. For other languages, such as Ruby, every chunk will start a new process and thus all data is lost between chunks, unless it is somehow stored in a data file that is read by the next chunk.

Being able to persist data between chunks is critical for literate programming otherwise the flow of the narrative is lost by all the effort of having to save data and then reload it. Although this might, at first, seem like a small nuisance, not being able to persist data between chunks is a major issue. For example, let’s take a look at the following simple example in which we want

to show how to create a list and the use it. Let's first assume that data cannot be persisted between chunks. In the next chunk we create a list, then we would need to save it to file, but to save it, we need somehow to marshal the data into a binary format:

```
lst = R.list(a: 1, b: 2, c: 3)
lst.saveRDS("lst.rds")
```

then, on the next chunk, where variable 'lst' is used, we need to read back it's value

```
lst = R.readRDS("lst.rds")
puts lst
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

Now, any single code has dozens of variables that we might want to use and reuse between chunks. Clearly, such an approach becomes quickly unmanageable. Probably, because of this problem, it is very rare to see any **R markdown** document in the Ruby community.

When variables can be used across chunks, then no overhead is needed:

```
lst = R.list(a: 1, b: 2, c: 3)
# any other code can be added here
```

```
puts lst
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

In the Python community, the same effort to have code and text in an integrated environment started around the first decade of the 2000s. In 2006 IPython 0.7.2 was released. In 2014, Fernando Pérez spun off the Jupyter project from IPython, creating a web-based interactive computation environment. Jupyter can now be used with many languages, including Ruby with the iruby gem (<https://github.com/SciRuby/iruby>). In order to have multiple languages in a Jupyter notebook the SoS kernel was developed (<https://vatlab.github.io/sos-docs/>).

9.1 gKnit and **R markdown**

gKnit is based on knitr and **R markdown** and can knit a document written both in Ruby and/or R and output it in any of the available formats of **R markdown**. gKnit allows ruby developers to do literate programming and reproducible research by allowing them to have in a single document, text and code.

In gKnit, Ruby variables are persisted between chunks, making it an ideal solution for literate programming in this language. Also, since it is based on Galaaz, Ruby chunks can access R variables (`~:name`, `R.*`) through the **Galaaz bridge** while knitr drives **GNU R**—no GraalVM polyglot runtime is required.

This is not a blog post on **R markdown**, and the interested user is directed to the following links for detailed information on its capabilities and use.

- <https://rmarkdown.rstudio.com/> or
- <https://bookdown.org/yihui/rmarkdown/>

In this post, we will describe just the main aspects of **R markdown**, so the user can start gKnitting Ruby and R documents quickly.

9.2 The Yaml header

An **R markdown** document should start with a Yaml header and be stored in a file with ‘.Rmd’ extension. This document has the following header for gKnitting an HTML document.

```
---
title: "How to do reproducible research in Ruby with gKnit"
author:
  - "Rodrigo Botafogo"
  - "Daniel Mossé - University of Pittsburgh"
tags: [Tech, Data Science, Ruby, R, JRuby, Galaaz]
date: "20/02/2019"
output:
  html_document:
    self_contained: true
    keep_md: true
  pdf_document:
    includes:
      in_header: ["../sty/galaaz.sty"]
    number_sections: yes
---
```

For more information on the options in the Yaml header, check [here](#).

9.3 Choosing the output format when calling gknit

Yes: you can select the render target on the **command line**. `bin/gknit` (or `gknit` on your `PATH`) forwards options to `rmarkdown::render` via `R::Rmarkdown.render`.

- **--output_format FORMAT** — name of the format, as in the YAML `output:` block. Examples:
 - **html_document** — HTML (often the default you list first under `output:`).
 - **pdf_document** — PDF (you need a working LaTeX setup, e.g. TinyTeX; see `bin/install-tinytex`).
 - **md_document**, **github_document**, or any other format defined in your YAML.
 - **all** — render **every** format declared under `output:` in the document (same idea as in R Markdown).

If you **omit** `--output_format`, gknit passes **NULL** for the format argument. In that case **rmarkdown** uses the **first** format listed under `output:` in the YAML (and if none is specified there, behavior follows the usual rmarkdown defaults, typically HTML).

Other useful flags:

- **--output_file NAME** — output file name (optional path; see also `--output_dir`).
- **--output_dir DIR** — directory for the rendered file (created if missing).
- **--bridge_timeout_sec / --callback_timeout_ms** — longer R or install steps (see elsewhere in this manual).

Examples (run from the directory where paths make sense, or use absolute paths):

```
bin/gknit blogs/manual/manual.Rmd
bin/gknit --output_format html_document blogs/manual/manual.Rmd
bin/gknit --output_format pdf_document blogs/manual/manual.Rmd
bin/gknit --output_format all blogs/manual/manual.Rmd
```

Use `gknit -h` for the full option list.

9.4 R Markdown formatting

Document formatting can be done with simple markups such as:

9.5 Headers

```
# Header 1
```

```
## Header 2
```

```
### Header 3
```

9.6 Lists

Unordered lists:

```
* Item 1
* Item 2
  + Item 2a
  + Item 2b
```


Ordered Lists

1. Item 1
2. Item 2
3. Item 3
 - + Item 3a
 - + Item 3b

For more R markdown formatting go to https://rmarkdown.rstudio.com/authoring_basics.html.

9.7 R chunks

Running and executing Ruby and R code is actually what really interests us is this blog. Inserting a code chunk is done by adding code in a block delimited by three back ticks followed by an open curly brace ('{') followed with the engine name (r, ruby, rb, include, ...), an any optional chunk_label and options, as shown below:

```
“““{engine_name [chunk_label], [chunk_options]}
“““
```

for instance, let's add an R chunk to the document labeled 'first_r_chunk'. This is a very simple code just to create a variable and print it out, as follows:

```
“““{r first_r_chunk}
vec <- c(1, 2, 3)
print(vec)
“““
```

If this block is added to an **R markdown** document and gKnitted the result will be:

```
vec <- c(1, 2, 3)
print(vec)
```

```
## [1] 1 2 3
```

Now let's say that we want to do some analysis in the code, but just print the result and not the code itself. For this, we need to add the option 'echo = FALSE'.

```
“““{r second_r_chunk, echo = FALSE}
vec2 <- c(10, 20, 30)
vec3 <- vec * vec2
print(vec3)
“““
```

Here is how this block will show up in the document. Observe that the code is not shown and we only see the execution result in a white box

```
## [1] 10 40 90
```

A description of the available chunk options can be found in <https://yihui.name/knitr/>.

Let's add another R chunk with a function definition. In this example, a vector 'r_vec' is created and a new function 'reduce_sum' is defined. The chunk specification is

```
```{r data_creation}
r_vec <- c(1, 2, 3, 4, 5)

reduce_sum <- function(...) {
 Reduce(sum, as.list(...))
}
```
```

and this is how it will look like once executed. From now on, to be concise in the presentation we will not show chunk definitions any longer.

```
r_vec <- c(1, 2, 3, 4, 5)

reduce_sum <- function(...) {
  Reduce(sum, as.list(...))
}
```

We can, possibly in another chunk, access the vector and call the function as follows:

```
print(r_vec)
```

```
## [1] 1 2 3 4 5
```

```
print(reduce_sum(r_vec))
```

```
## [1] 15
```

9.8 R Graphics with ggplot

In the following chunk, we create a bubble chart in R using ggplot and include it in this document. Note that there is no directive in the code to include the image, this occurs automatically. The 'mpg' dataframe is natively available to R and to Galaaaz as well.

For the reader not knowledgeable of ggplot, ggplot is a graphics library based on "the grammar of graphics" (Wilkinson 2005). The idea of the grammar of graphics is to build a graphics by adding layers to the plot. More information can be found in <https://towardsdatascience.com/a-comprehensive-guide-to-the-grammar-of-graphics-for-effective-visualization-of-multi-dimensional-1f92b4ed4149>.

In the plot below the 'mpg' dataset from base R is used. "The data concerns city-cycle fuel consumption in miles per gallon, to be predicted in terms of 3 multivalued discrete and 5 continuous attributes." (Quinlan, 1993)

First, the 'mpg' dataset is filtered to extract only cars from the following manufacturers: Audi, Ford, Honda, and Hyundai and stored in the 'mpg_select' variable. Then, the selected dataframe is passed to the ggplot function specifying in the aesthetic method (aes) that 'displacement' (displ) should be plotted in the 'x' axis and 'city mileage' should be on the 'y' axis. In the 'labs' layer we pass the 'title' and 'subtitle' for the plot. To the basic plot 'g', geom_jitter is added, that plots cars from the same manufacturers with the same color (col=manufacturer) and the size of the car point equal its high way consumption (size = hwy). Finally, a last layer is plotted containing a linear regression line (method = "lm") for every manufacturer.

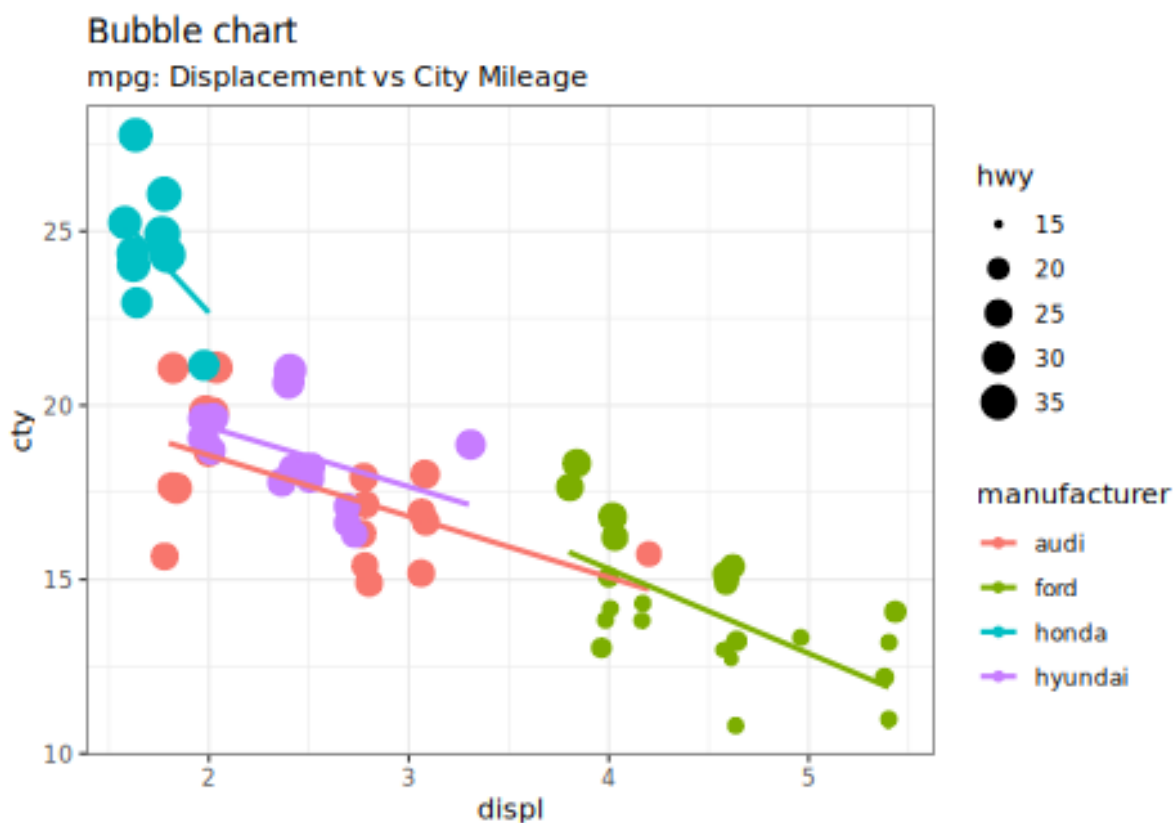
```
# load package and data
library(ggplot2)
data(mpg, package="ggplot2")

mpg_select <- mpg[mpg$manufacturer %in% c("audi", "ford", "honda", "hyundai"), ]

# Scatterplot
theme_set(theme_bw()) # pre-set the bw theme.
g <- ggplot(mpg_select, aes(displ, cty)) +
  labs(subtitle="mpg: Displacement vs City Mileage",
       title="Bubble chart")

g + geom_jitter(aes(col=manufacturer, size=hwy)) +
  geom_smooth(aes(col=manufacturer), method="lm", se=F)

## 'geom_smooth()' using formula = 'y ~ x'
```



9.9 Ruby chunks

Including a Ruby chunk is just as easy as including an R chunk in the document: just change the name of the engine to ‘ruby’. It is also possible to pass chunk options to the Ruby engine; however, this version does not accept all the options that are available to R chunks. Future versions will add those options.

```
‘‘‘{ruby first_ruby_chunk}
’’‘
```

In this example, the ruby chunk is called ‘first_ruby_chunk’. One important aspect of chunk labels is that they cannot be duplicated. If a chunk label is duplicated, gKnit will stop with an error.

In the following chunk, variable ‘a’, ‘b’ and ‘c’ are standard Ruby variables and ‘vec’ and ‘vec2’ are two vectors created by calling the ‘c’ method on the R module.

In Galaaz, the R module allows us to access R functions transparently. The ‘c’ function in R, is a function that concatenates its arguments making a vector.

It should be clear that there is no requirement in gknit to call or use any R functions. gKnit will knit standard Ruby code, or even general text without any code.

```
a = [1, 2, 3]
b = "US$ 250.000"
c = "The 'outputs' function"

vec = R.c(1, 2, 3)
vec2 = R.c(10, 20, 30)
```

In the next block, variables ‘a’, ‘vec’ and ‘vec2’ are used and printed.

```
puts a
puts vec * vec2
```

```
## 1
## 2
## 3
## [1] 10 40 90
```

Note that ‘a’ is a standard Ruby Array and ‘vec’ and ‘vec2’ are vectors that behave accordingly, where multiplication works as expected.

9.10 Inline Ruby code

When using a Ruby chunk, the code and the output are formatted in blocks as seen above. This formatting is not always desired. Sometimes, we want to have the results of the Ruby evaluation included in the middle of a phrase. gKnit allows adding inline Ruby code with the ‘rb’ engine. The following chunk specification will create and inline Ruby text:

This is some text with inline Ruby accessing variable 'b' which has value:

```
““{rb puts b}
““
```

and is followed by some other text!

This is some text with inline Ruby accessing variable 'b' which has value: US\$ 250.000 and is followed by some other text!

Note that it is important not to add any new line before or after the code block if we want everything to be in only one line, resulting in the following sentence with inline Ruby code.

9.10.1 The 'outputs' function

He have previously used the standard 'puts' method in Ruby chunks in order produce output. The result of a 'puts', as seen in all previous chunks that use it, is formatted inside a white box that follows the code block. Many times however, we would like to do some processing in the Ruby chunk and have the result of this processing generate and output that is "included" in the document as if we had typed it in **R markdown** document.

For example, suppose we want to create a new heading in our document, but the heading phrase is the result of some code processing: maybe it's the first line of a file we are going to read. Method 'outputs' adds its output as if typed in the **R markdown** document.

Take now a look at variable 'c' (it was defined in a previous block above) as 'c = "The 'outputs' function". "The 'outputs' function" is actually the name of this section and it was created using the 'outputs' function inside a Ruby chunk.

The ruby chunk to generate this heading is:

```
““{ruby heading}
outputs "### #{c}"
““
```

The three '###' is the way we add a Heading 3 in **R markdown**.

9.10.2 HTML Output from Ruby Chunks

We've just seen the use of method 'outputs' to add text to the the **R markdown** document. This technique can also be used to add HTML code to the document. In **R markdown**, any html code typed directly in the document will be properly rendered. Here, for instance, is a table definition in HTML and its output in the document:

```
<table style="width:100%">
  <tr>
    <th>Firstname</th>
    <th>Lastname</th>
    <th>Age</th>
  </tr>
  <tr>
    <td>Jill</td>
    <td>Smith</td>
    <td>50</td>
```

```

</tr>
<tr>
  <td>Eve</td>
  <td>Jackson</td>
  <td>94</td>
</tr>
</table>

```

Firstname

Lastname

Age

Jill

Smith

50

Eve

Jackson

94

But manually creating HTML output is not always easy or desirable, specially if we intend the document to be rendered in other formats, for example, as LaTeX. Also, The above table looks ugly. The ‘kableExtra’ library is a great library for creating beautiful tables. Take a look at https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_html.html

In the next chunk, we output the ‘mtcars’ dataframe from R in a nicely formatted table. Note that we retrieve the mtcars dataframe by using ‘~:mtcars’.

```

R.install_and_loads('kableExtra')
outputs (~:mtcars).kable.kable_styling

```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4

Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

9.11 Including Ruby files in a chunk

R is a language that was created to be easy and fast for statisticians to use. As far as I know, it was not a language to be used for developing large systems. Of course, there are large systems and libraries in R, but the focus of the language is for developing statistical models and distribute that to peers.

Ruby on the other hand, is a language for large software development. Systems written in Ruby will have dozens, hundreds or even thousands of files. To document a large system with literate programming, we cannot expect the developer to add all the files in a single ‘Rmd’ file. gKnit provides the ‘include’ chunk engine to include a Ruby file as if it had being typed in the ‘Rmd’ file.

To include a file, the following chunk should be created, where is the name of the file to be included and where the extension, if it is ‘rb’, does not need to be added. If the ‘relative’ option is not included, then it is treated as TRUE. When ‘relative’ is true, ruby’s ‘require_relative’ semantics is used to load the file, when false, Ruby’s \$LOAD_PATH is searched to find the file and it is ‘require’d.

```
“““{include <filename>, relative = <TRUE/FALSE>}
“““
```

Below we include file ‘model.rb’, which is in the same directory of this blog.

This code uses R ‘caret’ package to split a dataset in a train and test sets. The ‘caret’ package is a very important a useful package for doing Data Analysis, it has hundreds of functions for all steps of the Data Analysis workflow. To use ‘caret’ just to split a dataset is like using the proverbial cannon to kill the fly. We use it here only to show that integrating Ruby and R and using even a very complex package as ‘caret’ is trivial with Galaaz.

A word of advice: the ‘caret’ package has lots of dependencies and installing it in a Linux system is a time consuming operation. Method ‘R.install_and_loads’ will install the package if it is not already installed and can take a while.

```

'''{include model}
'''

require 'galaaz'

# Loads the R 'caret' package. If not present, installs it
R.install_and_loads 'caret'

class Model

  attr_reader :data
  attr_reader :test
  attr_reader :train

  #=====
  #
  #=====

  def initialize(data, percent_train:, seed: 123)

    R.set__seed(seed)
    @data = data
    @percent_train = percent_train
    @seed = seed

  end

  #=====
  #
  #=====

  def partition(field)

    train_index =
      R.createDataPartition(@data.send(field), p: @percent_train,
                           list: false, times: 1)
    @train = @data[train_index, :all]
    @test = @data[-train_index, :all]

  end

end
end

```

```

mtcars = ~:mtcars
model = Model.new(mtcars, percent_train: 0.8)
model.partition(:mpg)
puts model.train.head
puts model.test.head

```

```

##              mpg cyl  disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4    21.0   6 160.0 110  3.90 2.620 16.46  0  1   4     4

```



```
## Datsun 710      22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
##               mpg cyl  disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Duster 360     14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
```

9.12 Documenting Gems

gKnit also allows developers to document and load files that are not in the same directory of the ‘.Rmd’ file.

Here is an example of loading Ruby’s standard library file `find.rb`. In this example, `relative` is set to `FALSE`, so Ruby will look for the file in its `$LOAD_PATH`, and the user does not need to know its directory on disk.

```
‘‘‘{include find, relative = FALSE}
‘‘‘
```

```
# frozen_string_literal: true
#
# find.rb: the Find module for processing all files under a given directory.
#
#
# The +Find+ module supports the top-down traversal of a set of file paths.
#
# For example, to total the size of all files under your home directory,
# ignoring anything in a "dot" directory (e.g. $HOME/.ssh):
#
#   require 'find'
#
#   total_size = 0
#
#   Find.find(ENV["HOME"]) do |path|
#     if FileTest.directory?(path)
#       if File.basename(path).start_with?('.')
#         Find.prune      # Don't look any further into this directory.
#       else
#         next
#       end
#     else
#       total_size += FileTest.size(path)
#     end
#   end
#
module Find
```

```

VERSION = "0.2.0"

#
# Calls the associated block with the name of every file and directory listed
# as arguments, then recursively on their subdirectories, and so on.
#
# Returns an enumerator if no block is given.
#
# See the +Find+ module documentation for an example.
#
def find(*paths, ignore_error: true) # :yield: path
  block_given? or return enum_for(__method__, *paths, ignore_error: ignore_error)

  fs_encoding = Encoding.find("filesystem")

  paths.collect!{|d| raise Errno::ENOENT, d unless File.exist?(d); d.dup}.each do |path|
    path = path.to_path if path.respond_to? :to_path
    enc = path.encoding == Encoding::US_ASCII ? fs_encoding : path.encoding
    ps = [path]
    while file = ps.shift
      catch(:prune) do
        yield file.dup
        begin
          s = File.lstat(file)
          rescue Errno::ENOENT, Errno::EACCES, Errno::ENOTDIR, Errno::ELOOP, Errno::ENAMETOOLONG
            raise unless ignore_error
          next
        end
        if s.directory? then
          begin
            fs = Dir.children(file, encoding: enc)
            rescue Errno::ENOENT, Errno::EACCES, Errno::ENOTDIR, Errno::ELOOP, Errno::ENAMETOOLONG
              raise unless ignore_error
            next
          end
          fs.sort!
          fs.reverse_each {|f|
            f = File.join(file, f)
            ps.unshift f
          }
        end
      end
    end
  end
  nil
end

#
# Skips the current file or directory, restarting the loop with the next
# entry. If the current file is a directory, that directory will not be

```

```
# recursively entered. Meaningful only within the block associated with
# Find::find.
#
# See the +Find+ module documentation for an example.
#
def prune
  throw :prune
end

module_function :find, :prune
end
```

9.13 Converting to PDF

One of the beauties of knitr is that the same input can be converted to many different outputs. One very useful format, is, of course, PDF. In order to converted an **R markdown** file to PDF it is necessary to have LaTeX installed on the system. We will not explain here how to install LaTeX as there are plenty of documents on the web showing how to proceed.

gKnit comes with a simple LaTeX style file for gknitting this blog as a PDF document. Here is the Yaml header to generate this blog in PDF format instead of HTML:

```
---
title: "gKnit - Ruby and R Knitting with Galaaz"
author: "Rodrigo Botafogo"
tags: [Galaaz, Ruby, R, JRuby, knitr, gknit]
date: "29 October 2018"
output:
  pdf\_document:
    includes:
      in\_header: ["../sty/galaaz.sty"]
    number\_sections: yes
---
```

9.14 Template based documents generation

When a document is converted to PDF it follows a certain conversion template. We've seen above the use of 'galaaz.sty' as a basic template to generate a PDF document. Using the 'gknit-draft' app that comes with Galaaz, the same .Rmd file can be compiled to different looking PDF documents. Galaaz automatically loads the 'rticles' R package that comes with templates for the following journals with the respective template name:

- ACM articles: acm_article
- ACS articles: acs_article
- AEA journal submissions: aea_article
- AGU journal submissions: ????
- AMS articles: ams_article
- American Statistical Association: asa_article
- Biometrics articles: biometrics_article
- Bulletin de l'AMQ journal submissions: amq_article

- CTeX documents: `ctex`
- Elsevier journal submissions: `elsevier_article`
- IEEE Transaction journal submissions: `ieee_article`
- JSS articles: `jss_article`
- MDPI journal submissions: `mdpi_article`
- Monthly Notices of the Royal Astronomical Society articles: `mnras_article`
- NNRAS journal submissions: `nrmnas_article`
- PeerJ articles: `peerj_article`
- Royal Society Open Science journal submissions: `rsos_article`
- Royal Statistical Society: `rss_article`
- Sage journal submissions: `sage_article`
- Springer journal submissions: `springer_article`
- Statistics in Medicine journal submissions: `sim_article`
- Copernicus Publications journal submissions: `copernicus_article`
- The R Journal articles: `rjournal_article`
- Frontiers articles: ???
- Taylor & Francis articles: ???
- Bulletin De L'AMQ: `amq_article`
- PLOS journal: `plos_article`
- Proceedings of the National Academy of Sciences of the USA: `pnas_article`

In order to create a document with one of those templates, use the following command:

```
gknit-draft --filename <my_document> --template <template> --package <package>
--create_dir
```

So, in order to create a template for writing an R Journal, use:

```
gknit-draft --filename my_r_article --template rjournal_article --package rticles
--create_dir
```

10 Accessing R variables

Galaaz allows Ruby to access variables created in R. For example, the `mtcars` data set is available in R and can be accessed from Ruby by using the tilde operator followed by the symbol for the variable, in this case `:mtcars`. In the code below, method `outputs` is used to output the `mtcars` data set nicely formatted in HTML by use of the `kable` and `kable_styling` functions. Method `outputs` is only available when used with gKnit.

```
outputs (~:mtcars).kable.kable_styling
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

11 Basic Data Types

11.1 Vector

Vectors can be thought of as contiguous cells containing data. Cells are accessed through indexing operations such as `x[5]`. Galaaz has six basic ('atomic') vector types: logical, integer, real, complex, string (or character) and raw. The modes and storage modes for the different vector types are listed in the following table.

typeof	mode	storage.mode
logical	logical	logical
integer	numeric	integer
double	numeric	double
complex	complex	complex
character	character	character
raw	raw	raw

Single numbers, such as 4.2, and strings, such as "four point two" are still vectors, of length 1; there are no more basic types. Vectors with length zero are possible (and useful). String

vectors have mode and storage mode “character”. A single element of a character vector is often referred to as a character string.

To create a vector the ‘c’ (concatenate) method from the ‘R’ module should be used:

```
vec = R.c(1, 2, 3)
puts vec
```

```
## [1] 1 2 3
```

Lets take a look at the type, mode and storage.mode of our vector vec. In order to print this out, we are creating a data frame ‘df’ and printing it out. A data frame, for those not familiar with it, is basically a table. Here we create the data frame and add the column name by passing named parameters for each column, such as ‘typeof:’, ‘mode:’ and ‘storage__mode:’. You should also note here that the double underscore is converted to a ‘.’. So, when printed ‘storage__mode’ will actually print as ‘storage.mode’.

Data frames will later be more carefully described. In R, the method used to create a data frame is ‘data.frame’, in Galaaz we use ‘data__frame’.

```
df = R.data__frame(typeof: vec.typeof, mode: vec.mode, storage__mode: vec.storage__mode)
puts df
```

```
##      typeof      mode storage.mode
## 1 integer numeric          integer
```

If you want to create a vector with floating point numbers, then we need at least one of the vector’s element to be a float, such as 1.0. R users should be careful, since in R a number like ‘1’ is converted to float and to have an integer the R developer will use ‘1L’. Galaaz follows normal Ruby rules and the number 1 is an integer and 1.0 is a float.

```
vec = R.c(1.0, 2, 3)
puts vec
```

```
## [1] 1 2 3
```

```
df = R.data__frame(typeof: vec.typeof, mode: vec.mode, storage__mode: vec.storage__mode)
outputs df.kable.kable_styling
```

typeof	mode	storage.mode
double	numeric	double

In this next example we try to create a vector with a variable ‘hello’ that has not yet being defined. This will raise an exception that is printed out. We get two return blocks, the first with a message explaining what went wrong and the second with the full backtrace of the error.

```
vec = R.c(1, hello, 5)
```

```
## undefined local variable or method 'hello' for an instance of RC
```

```
## /home/rbotafogo/desenv_linux/galaaz/lib/util/exec_ruby.rb:170:in 'exec_ruby'
## org/jruby/RubyKernel.java:1268:in 'eval'
## /home/rbotafogo/desenv_linux/galaaz/lib/util/exec_ruby.rb:169:in 'exec_ruby'
## /home/rbotafogo/desenv_linux/galaaz/lib/gknit/knitr_engine.rb:777:in 'block in initialize'
## org/jruby/RubyBasicObject.java:2695:in 'instance_eval'
## org/jruby/RubyBasicObject.java:2723:in 'instance_eval'
## /home/rbotafogo/desenv_linux/galaaz/lib/gknit/knitr_engine.rb:748:in 'block in initialize'
## /home/rbotafogo/desenv_linux/galaaz/lib/R_interface/new_bridge_adapter.rb:358:in 'block in initialize'
## /home/rbotafogo/desenv_linux/galaaz/lib/new_bridge/session_client.rb:413:in 'block in handle_request'
```

Here is a vector with logical values

```
vec = R.c(true, true, false, false, true)
puts vec
```

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

11.1.1 Combining Vectors

The 'c' functions used to create vectors can also be used to combine two vectors:

```
vec1 = R.c(10.0, 20.0, 30.0)
vec2 = R.c(4.0, 5.0, 6.0)
vec = R.c(vec1, vec2)
puts vec
```

```
## [1] 10 20 30 4 5 6
```

In galaaz, methods can be chained (somewhat like the pipe operator in R %>%, but more generic). In this next example, method 'c' is chained after 'vec1'. This also looks like 'c' is a method of the vector, but in reality, this is actually closer to the pipe operator. When Galaaz identifies that 'c' is not a method of 'vec' it actually tries to call 'R.c' with 'vec1' as the first argument concatenated with all the other available arguments. The code below is automatically converted to the code above.

```
vec = vec1.c(vec2)
puts vec
```

```
## [1] 10 20 30 4 5 6
```

11.1.2 Vector Arithmetic

Arithmetic operations on vectors are performed element by element:

```
puts vec1 + vec2
```

```
## [1] 14 25 36
```

```
puts vec1 * 5
```

```
## [1] 50 100 150
```

When vectors have different length, a recycling rule is applied to the shorter vector:

```
vec3 = R.c(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
```

```
puts vec4 = vec1 + vec3
```

```
## [1] 11 22 33 14 25 36 17 28 39
```

11.1.3 Vector Indexing

Vectors can be indexed by using the ‘[]’ operator:

```
puts vec4[3]
```

```
## [1] 33
```

We can also index a vector with another vector. For example, in the code below, we take elements 1, 3, 5, and 7 from vec3:

```
puts vec4[R.c(1, 3, 5, 7)]
```

```
## [1] 11 33 25 17
```

Repeating an index and having indices out of order is valid code:

```
puts vec4[R.c(1, 3, 3, 1)]
```

```
## [1] 11 33 33 11
```

It is also possible to index a vector with a negative number or negative vector. In these cases the indexed values are not returned:

```
puts vec4[-3]
```

```
puts vec4[-R.c(1, 3, 5, 7)]
```

```
## [1] 11 22 14 25 36 17 28 39
```

```
## [1] 22 14 36 28 39
```

If an index is out of range, a missing value (NA) will be reported.


```
puts vec4[30]
```

```
## [1] NA
```

It is also possible to index a vector by range:

```
puts vec4[(2..5)]
```

```
## [1] 22 33 14 25
```

Elements in a vector can be named using the ‘names’ attribute of a vector:

```
full_name = R.c("Rodrigo", "A", "Botafogo")
full_name.names = R.c("First", "Middle", "Last")
puts full_name
```

```
##      First      Middle      Last
## "Rodrigo"      "A" "Botafogo"
```

Or it can also be named by using the ‘c’ function with named parameters:

```
full_name = R.c(First: "Rodrigo", Middle: "A", Last: "Botafogo")
puts full_name
```

```
##      First      Middle      Last
## "Rodrigo"      "A" "Botafogo"
```

11.1.4 Extracting Native Ruby Types from a Vector

Vectors created with ‘R.c’ are of class R::Vector. You might have noticed that when indexing a vector, a new vector is returned, even if this vector has one single element. In order to use R::Vector with other ruby classes it might be necessary to extract the actual Ruby native type from the vector. In order to do this extraction the ‘»’ operator is used.

```
puts vec4
puts vec4 >> 0
puts vec4 >> 4
```

```
## [1] 11 22 33 14 25 36 17 28 39
## 11.0
## 25.0
```

Note that indexing with ‘»’ starts at 0 and not at 1, also, we cannot do negative indexing.

11.2 Matrix

A matrix is a collection of elements organized as a two dimensional table. A matrix can be created by the ‘matrix’ function:

```
mat = R.matrix(R.c(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0),
               nrow: 3,
               ncol: 3)

puts mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Note that matrices data is organized by column first. It is possible to organize the matrix memory by row first passing an extra argument to the ‘matrix’ function:

```
mat_row = R.matrix(R.c(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0),
                   nrow: 3,
                   ncol: 3,
                   byrow: true)

puts mat_row
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

11.2.1 Indexing a Matrix

A matrix can be indexed by [row, column]:

```
puts mat_row[1, 1]
puts mat_row[2, 3]
```

```
## [1] 1
## [1] 6
```

It is possible to index an entire row or column with the ‘:all’ keyword

```
puts mat_row[1, :all]
puts mat_row[:all, 2]
```

```
## [1] 1 2 3
## [1] 2 5 8
```

Indexing with a vector is also possible for matrices. In the following example we want rows 1 and 3 and columns 2 and 3 building a 2 x 2 matrix.

```
puts mat_row[R.c(1, 3), R.c(2, 3)]
```

```
##      [,1] [,2]
## [1,]    2    3
## [2,]    8    9
```

Matrices can be combined with functions ‘rbind’:

```
puts mat_row.rbind(mat)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]    1    4    7
## [5,]    2    5    8
## [6,]    3    6    9
```

and ‘cbind’:

```
puts mat_row.cbind(mat)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    1    4    7
## [2,]    4    5    6    2    5    8
## [3,]    7    8    9    3    6    9
```

11.3 List

A list is a data structure that can contain sublists of different types, while vector and matrix can only hold one type of element.

```
nums = R.c(1.0, 2.0, 3.0)
strs = R.c("a", "b", "c", "d")
bool = R.c(true, true, false)
lst = R.list(nums: nums, strs: strs, bool: bool)
puts lst
```

```
## $nums
## [1] 1 2 3
##
## $strs
## [1] "a" "b" "c" "d"
##
## $bool
## [1] TRUE TRUE FALSE
```

Note that ‘lst’ elements are named elements.

11.3.1 List Indexing

List indexing, also called slicing, is done using the ‘[]’ operator and the ‘[[[]]’ operator. Let’s first start with the ‘[]’ operator. The list above has three sublist indexing with ‘[]’ will return one of the sublists.

```
puts lst[1]
```

```
## $nums
## [1] 1 2 3
```

Note that when using ‘[]’ a new list is returned. When using the double square bracket operator the value returned is the actual element of the list in the given position and not a slice of the original list

```
puts lst[[1]]
```

```
## [1] 1 2 3
```

When elements are named, as done with `lst`, indexing can be done by name:

```
puts lst[['bool']][[1]] >> 0
```

```
## true
```

In this example, first the ‘bool’ element of the list was extracted, not as a list, but as a vector, then the first element of the vector was extracted (note that vectors also accept the ‘[[[]]’ operator) and then the vector was indexed by its first element, extracting the native Ruby type.

11.4 Data Frame

A data frame is a table like structure in which each column has the same number of rows. Data frames are the basic structure for storing data for data analysis. We have already seen a data frame previously when we accessed variable ‘`~:mtcars`’. In order to create a data frame, function ‘`data__frame`’ is used:

```
df = R.data__frame(
  year: R.c(2010, 2011, 2012),
  income: R.c(1000.0, 1500.0, 2000.0))
```

```
puts df
```

```
##   year income
## 1 2010   1000
## 2 2011   1500
## 3 2012   2000
```

11.4.1 Data Frame Indexing

A data frame can be indexed the same way as a matrix, by using ‘[row, column]’, where row and column can either be a numeric or the name of the row or column

```
puts (~:mtcars).head
puts (~:mtcars)[1, 2]
puts (~:mtcars)['Datsun 710', 'mpg']
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4      21.0   6  160 110 3.90 2.620 16.46 0  1   4    4
## Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02 0  1   4    4
## Datsun 710      22.8   4  108  93 3.85 2.320 18.61 1  1   4    1
## Hornet 4 Drive  21.4   6  258 110 3.08 3.215 19.44 1  0   3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02 0  0   3    2
## Valiant        18.1   6  225 105 2.76 3.460 20.22 1  0   3    1
## [1] 6
## [1] 22.8
```

Extracting a column from a data frame as a vector can be done by using the double square bracket operator:

```
puts (~:mtcars)[['mpg']]
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

A data frame column can also be accessed as if it were an instance variable of the data frame:

```
puts (~:mtcars).mpg
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

Slicing a data frame can be done by indexing it with a vector (we use ‘head’ to reduce the output):

```
puts (~:mtcars)[R.c('mpg', 'hp')].head
```

```
##      mpg cyl disp hp drat wt  qsec vs am gear carb
## NA     NA  NA   NA NA   NA NA   NA NA NA   NA   NA
## NA.1   NA  NA   NA NA   NA NA   NA NA NA   NA   NA
```

A row slice can be obtained by indexing by row and using the ‘:all’ keyword for the column:

```
puts (~:mtcars)[R.c('Datsun 710', 'Camaro Z28'), :all]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Datsun 710 22.8   4  108  93 3.85 2.32 18.61  1  1   4    1
## Camaro Z28 13.3   8  350 245 3.73 3.84 15.41  0  0   3    4
```

Finally, a data frame can also be indexed with a logical vector. In this next example, the ‘am’ column of :mtcars is compared with 0 (with method ‘eq’). When ‘am’ is equal to 0 the car is automatic. So, by doing ‘(~:mtcars).am.eq 0’ a logical vector is created with ‘true’ whenever ‘am’ is 0 and ‘false’ otherwise.

```
# obtain a vector with 'true' for cars with automatic transmission
automatic = (~:mtcars).am.eq 0
puts automatic
```

```
## [1] FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [13] TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE TRUE TRUE TRUE TRUE
## [25] TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Using this logical vector, the data frame is indexed, returning a new data frame in which all cars have automatic transmission.

```
# slice the data frame by using this vector
puts (~:mtcars)[automatic, :all]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0   3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0   3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0   3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0   3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0   4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0   4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0   4    4
## Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0   4    4
## Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0   3    3
## Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0   3    3
## Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0   3    3
## Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98  0  0   3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0   3    4
## Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0  0   3    4
## Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0   3    1
## Dodge Challenger  15.5   8 318.0 150 2.76 3.520 16.87  0  0   3    2
## AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0   3    2
## Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0   3    4
## Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0   3    2
```

12 Writing Expressions in Galaaz

Galaaz extends Ruby to work with complex expressions, similar to R’s expressions build with ‘quote’ (base R) or ‘quo’ (tidyverse). Let’s take a look at some of those expressions.

12.1 Expressions from operators

The code below creates an expression summing two symbols

```
exp1 = :a + :b
puts exp1
```

```
## a + b
```

We can build any complex mathematical expression

```
exp2 = (:a + :b) * 2.0 + :c ** 2 / :z
puts exp2
```

```
## a + b * 2.0 + c ^ 2L / z
```

It is also possible to use inequality operators in building expressions

```
exp3 = (:a + :b) >= :z
puts exp3
```

```
## a + b >= z
```

Galaaz provides both symbolic representations for operators, such as ($>$, $<$, $!=$) as functional notation for those operators such as (`.gt`, `.ge`, etc.). So the same expression written above can also be written as

```
exp4 = (:a + :b).ge :z
puts exp4
```

```
## a + b >= z
```

Two type of expression can only be created with the functional representation of the operators, those are expressions involving `'=='`, and `'='`. In order to write an expression involving `'=='` we need to use the method `'eq'` and for `'='` we need the function `'assign'`

```
exp5 = (:a + :b).eq :z
puts exp5
```

```
## a + b == z
```

```
exp6 = :y.assign :a + :b
puts exp6
```

```
## y <- a + b
```

In general we think that using the functional notation is preferable to using the symbolic notation as otherwise, we end up writing invalid expressions such as

```
exp_wrong = (:a + :b) == :z
puts exp_wrong
```

and it might be difficult to understand what is going on here. The problem lies with the fact that when using ‘==’ we are comparing expression (:a + :b) to expression :z with ‘==’. When the comparison is executed, the system tries to evaluate :a, :b and :z, and those symbols at this time are not bound to anything and we get a “object ‘a’ not found” message. If we only use functional notation, this type of error will not occur.

12.2 Expressions with R methods

It is often necessary to create an expression that uses a method or function. For instance, in mathematics, it’s quite natural to write an expression such as $y = \sin(x)$. In this case, the ‘sin’ function is part of the expression and should not immediately be executed. Now, let’s say that ‘x’ is an angle of 45° and we actually want our expression to be $y = 0.850\dots$. When we want the function to be part of the expression, we call the function preceding it by the letter E, such as ‘E.sin(x)’

```
exp7 = :y.assign E.sin(:x)
puts exp7
```

```
## y <- sin(x)
```

Expressions can also be written using ‘.’ notation:

```
exp8 = :y.assign :x.sin
puts exp8
```

```
## y <- sin(x)
```

When a function has multiple arguments, the first one can be used before the ‘:’:

```
exp9 = :x.c(:y)
puts exp9
```

```
## c(x, y)
```

12.3 Evaluating an Expression

Expressions can be evaluated by calling function ‘eval’ with a binding. A binding can be provided with a list:

```
exp = (:a + :b) * 2.0 + :c ** 2 / :z
puts exp.eval(R.list(a: 10, b: 20, c: 30, z: 40))
```

```
## [1] 72.5
```

... with a data frame:


```
df = R.data__frame(
  a: R.c(1, 2, 3),
  b: R.c(10, 20, 30),
  c: R.c(100, 200, 300),
  z: R.c(1000, 2000, 3000))
```

```
puts exp.eval(df)
```

```
## [1] 31 62 93
```

13 Manipulating Data

One of the major benefits of Galaaz is to bring strong data manipulation to Ruby. The following examples were extracted from Hadley’s “R for Data Science” (<https://r4ds.had.co.nz/>). This is a highly recommended book for those not already familiar with the ‘tidyverse’ style of programming in R. In the sections to follow, we will limit ourselves to convert the R code to Galaaz.

For these examples, we will investigate the `nycflights13` data set available on the package by the same name. We use function ‘`R.install_and_loads`’ that checks if the library is available locally, and if not, installs it. This data frame contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics.

Dplyr often uses **tibbles** in place of classic data frames. In Galaaz, printing may differ from the R console; if you need a classic tabular printout, convert with `as__data__frame` (or use `head` / `str` in R via R calls).

```
R.install_and_loads('nycflights13')
R.library('dplyr')
```

```
flights = ~:flights
puts flights.head
```

```
## # A tibble: 6 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
## 1  2013     1     1     517           515           2     830           819
## 2  2013     1     1     533           529           4     850           830
## 3  2013     1     1     542           540           2     923           850
## 4  2013     1     1     544           545          -1    1004          1022
## 5  2013     1     1     554           600          -6     812           837
## 6  2013     1     1     554           558          -4     740           728
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

13.1 Filtering rows with Filter

In this example we filter the flights data set by giving to the filter function two expressions: the first `:month.eq 1`

```
puts flights.filter((:month.eq 1), (:day.eq 1)).head
```

```
## # A tibble: 6 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>       <dbl>   <int>         <int>
## 1  2013     1     1     517           515         2     830           819
## 2  2013     1     1     533           529         4     850           830
## 3  2013     1     1     542           540         2     923           850
## 4  2013     1     1     544           545        -1    1004          1022
## 5  2013     1     1     554           600        -6     812           837
## 6  2013     1     1     554           558        -4     740           728
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

13.2 Logical Operators

All flights that departed in November or December

```
puts flights.filter((:month.eq 11) | (:month.eq 12)).head
```

```
## # A tibble: 6 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>       <dbl>   <int>         <int>
## 1  2013    11     1         5          2359         6     352           345
## 2  2013    11     1        35          2250       105     123           2356
## 3  2013    11     1       455           500        -5     641           651
## 4  2013    11     1       539           545        -6     856           827
## 5  2013    11     1       542           545        -3     831           855
## 6  2013    11     1       549           600       -11     912           923
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

The same as above, but using the ‘in’ operator. In R, it is possible to define many operators by doing `%%`. The `%in%` operator checks if a value is in a vector. In order to use those operators from `Galaxy` the `‘_’` method is used, where the first argument is the operator’s symbol, in this case `‘in’` and the second argument is the vector:

```
puts flights.filter(:month._ :in, R.c(11, 12)).head
```

```
## # A tibble: 6 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>       <dbl>   <int>         <int>
## 1  2013    11     1         5          2359         6     352           345
## 2  2013    11     1        35          2250       105     123           2356
## 3  2013    11     1       455           500        -5     641           651
## 4  2013    11     1       539           545        -6     856           827
```

```
## 5 2013    11    1    542          545      -3    831          855
## 6 2013    11    1    549          600     -11    912          923
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

13.3 Filtering with NA (Not Available)

Let's first create a 'tibble' with a Not Available value (R::NA). Tibbles are a modern version of a data frame and operate very similarly to one. It differs in how it outputs the values and the result of some subsetting operations that are more consistent than what is obtained from data frame.

```
df = R.tibble(x: R.c(1, R::NA, 3))
puts df
```

```
## # A tibble: 3 x 1
##       x
##   <int>
## 1     1
## 2    NA
## 3     3
```

Now filtering by `x > 1` shows all lines that satisfy this condition, where the row with R:NA does not.

```
puts df.filter(x > 1)
```

```
## # A tibble: 1 x 1
##       x
##   <int>
## 1     3
```

To match an NA use method 'is__na'

```
puts df.filter((x.is__na) | (x > 1))
```

```
## # A tibble: 2 x 1
##       x
##   <int>
## 1    NA
## 2     3
```

13.4 Arrange Rows with arrange

Arrange reorders the rows of a data frame by the given arguments.

```
puts flights.arrange(:year, :month, :day).head
```

```
## # A tibble: 6 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>       <dbl>   <int>         <int>
## 1  2013     1     1     517           515         2     830           819
## 2  2013     1     1     533           529         4     850           830
## 3  2013     1     1     542           540         2     923           850
## 4  2013     1     1     544           545        -1    1004          1022
## 5  2013     1     1     554           600        -6     812           837
## 6  2013     1     1     554           558        -4     740           728
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

To arrange in descending order, use function ‘desc’

```
puts flights.arrange(:dep_delay.desc).head
```

```
## # A tibble: 6 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>       <dbl>   <int>         <int>
## 1  2013     1     9     641           900      1301    1242          1530
## 2  2013     6    15    1432          1935      1137    1607          2120
## 3  2013     1    10    1121          1635      1126    1239          1810
## 4  2013     9    20    1139          1845      1014    1457          2210
## 5  2013     7    22     845          1600      1005    1044          1815
## 6  2013     4    10    1100          1900       960    1342          2211
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

13.5 Selecting columns

To select specific columns from a dataset we use function ‘select’:

```
puts flights.select(:year, :month, :day).head
```

```
## # A tibble: 6 x 3
##   year month   day
##   <int> <int> <int>
## 1  2013     1     1
## 2  2013     1     1
## 3  2013     1     1
## 4  2013     1     1
## 5  2013     1     1
## 6  2013     1     1
```

It is also possible to select column in a given range

```
puts flights.select(:year.up_to :day).head
```

```
## # A tibble: 6 x 3
##   year month   day
##   <int> <int> <int>
## 1  2013     1     1
## 2  2013     1     1
## 3  2013     1     1
## 4  2013     1     1
## 5  2013     1     1
## 6  2013     1     1
```

Select all columns that start with a given name sequence

```
puts flights.select(E.starts_with('arr')).head
```

```
## # A tibble: 6 x 2
##   arr_time arr_delay
##   <int>      <dbl>
## 1     830         11
## 2     850         20
## 3     923         33
## 4    1004        -18
## 5     812        -25
## 6     740         12
```

Other functions that can be used:

- `ends_with("xyz")`: matches names that end with “xyz”.
- `contains("ijk")`: matches names that contain “ijk”.
- `matches("(.)\\1")`: selects variables that match a regular expression. This one matches any variables that contain repeated characters.
- `num_range("x", (1..3))`: matches x1, x2 and x3

A helper function that comes in handy when we just want to rearrange column order is ‘Everything’:

```
puts flights.select(:year, :month, :day, E.everything).head
```

```
## # A tibble: 6 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>      <dbl>    <int>         <int>
## 1  2013     1     1     517           515         2      830           819
## 2  2013     1     1     533           529         4      850           830
## 3  2013     1     1     542           540         2      923           850
```

```
## 4 2013      1      1      544      545      -1      1004      1022
## 5 2013      1      1      554      600      -6       812       837
## 6 2013      1      1      554      558      -4       740       728
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

13.6 Add variables to a dataframe with 'mutate'

```
flights_sm = flights.
  select( (:year.up_to :day),
          E.ends_with('delay'),
          :distance,
          :air_time)

puts flights_sm.head
```

```
## # A tibble: 6 x 7
##   year month   day dep_delay arr_delay distance air_time
##   <int> <int> <int>     <dbl>     <dbl>     <dbl>     <dbl>
## 1  2013     1     1         2         11      1400       227
## 2  2013     1     1         4         20      1416       227
## 3  2013     1     1         2         33      1089       160
## 4  2013     1     1        -1        -18      1576       183
## 5  2013     1     1        -6        -25       762       116
## 6  2013     1     1        -4         12       719       150
```

```
flights_sm = flights_sm.
  mutate(gain: :dep_delay - :arr_delay,
         speed: :distance / :air_time * 60)

puts flights_sm.head
```

```
## # A tibble: 6 x 9
##   year month   day dep_delay arr_delay distance air_time   gain speed
##   <int> <int> <int>     <dbl>     <dbl>     <dbl>     <dbl> <dbl> <dbl>
## 1  2013     1     1         2         11      1400       227    -9  370.
## 2  2013     1     1         4         20      1416       227   -16  374.
## 3  2013     1     1         2         33      1089       160  -31  408.
## 4  2013     1     1        -1        -18      1576       183    17  517.
## 5  2013     1     1        -6        -25       762       116    19  394.
## 6  2013     1     1        -4         12       719       150   -16  288.
```

13.7 Summarising data

Function 'summarise' calculates summaries for the data frame. When no 'group_by' is used a single value is obtained from the data frame:

```
puts flights.summarise(delay: E.mean(:dep_delay, na_rm: true))

## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1  12.6
```

When a data frame is grouped with 'group_by' summaries apply to the given group:

```
by_day = flights.group_by(:year, :month, :day)
puts by_day.summarise(delay: :dep_delay.mean(na_rm: true)).head

## # A tibble: 6 x 4
## # Groups:   year, month [1]
##   year month   day delay
##   <int> <int> <int> <dbl>
## 1  2013     1     1  11.5
## 2  2013     1     2  13.9
## 3  2013     1     3  11.0
## 4  2013     1     4   8.95
## 5  2013     1     5   5.73
## 6  2013     1     6   7.15
```

Next we put many operations together by pipping them one after the other:

```
delays = flights.
  group_by(:dest).
  summarise(
    count: E.n,
    dist: :distance.mean(na_rm: true),
    delay: :arr_delay.mean(na_rm: true)).
  filter(:count > 20, :dest != "NHL")

puts delays.head

## # A tibble: 6 x 4
##   dest  count  dist delay
##   <chr> <int> <dbl> <dbl>
## 1 ABQ    254  1826   4.38
## 2 ACK    265   199   4.85
## 3 ALB    439   143  14.4
## 4 ATL  17215   757  11.3
## 5 AUS   2439  1514   6.02
## 6 AVL    275   584   8.00
```

14 Using Data Table

The next chunk reads the **flights14** sample (same file as the data.table vignette). Prefer downloading in Ruby and R.fread on a local path: `data.table::fread("https://...")`

runs the HTTP transfer **inside R** while the Galaaz bridge waits on a **single** eval; if the remote server stalls, that looks like an intermittent “hang” and eventually hits the **per-eval** bridge timeout. Ruby’s `Net::HTTP` `open_timeout` / `read_timeout` fail fast with a clear error instead. The chunk passes `nrows:` to `fread` so a single bridge eval does not have to materialize the full ~1.2M-row table during a gknit run.

```
require 'net/http'
require 'uri'
require 'tmpdir'

R.library('data.table')

url = URI('https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights.csv')
csv = File.join(Dir.tmpdir, "galaaz_manual_flights14_#{Process.pid}.csv")
begin
  Net::HTTP.start(url.host, url.port, use_ssl: true, open_timeout: 25, read_timeout: 120) do |http|
    resp = http.request_get(url.request_uri)
    raise "HTTP #{resp.code}" unless resp.is_a?(Net::HTTPSuccess)

    File.binwrite(csv, resp.body)
  end

  # Cap rows so one bridge eval stays short on slow disks (full file is ~1.2M rows).
  flights = R.fread(csv, nrows: 120_000)
  puts flights.dim
  puts R.head(flights, 12)
ensure
  File.unlink(csv) if csv && File.exist?(csv)
end
```

```
## [1] 120000      11
##      year month   day dep_delay arr_delay carrier origin  dest air_time
##      <int> <int> <int>      <int>      <int>  <char> <char> <char>    <int>
##  1:  2014     1     1        14         13     AA   JFK   LAX      359
##  2:  2014     1     1         -3         13     AA   JFK   LAX      363
##  3:  2014     1     1          2          9     AA   JFK   LAX      351
##  4:  2014     1     1         -8        -26     AA   LGA   PBI      157
##  5:  2014     1     1          2          1     AA   JFK   LAX      350
##  6:  2014     1     1          4          0     AA   EWR   LAX      339
##  7:  2014     1     1         -2        -18     AA   JFK   LAX      338
##  8:  2014     1     1         -3        -14     AA   JFK   LAX      356
##  9:  2014     1     1         -1        -17     AA   JFK   MIA      161
## 10:  2014     1     1         -2        -14     AA   JFK   SEA      349
## 11:  2014     1     1         -5        -17     AA   EWR   MIA      161
## 12:  2014     1     1          7         -5     AA   JFK   SFO      365
##      distance  hour
##      <int> <int>
##  1:    2475     9
##  2:    2475    11
##  3:    2475    19
##  4:    1035     7
```



```
## 5:      2475      13
## 6:      2454      18
## 7:      2475      21
## 8:      2475      15
## 9:      1089      15
## 10:     2422      18
## 11:     1085      16
## 12:     2586      17
```

```
data_table = R.data_table(
  ID: R.c("b", "b", "b", "a", "a", "c"),
  a: (1..6),
  b: (7..12),
  c: (13..18)
)

puts data_table
puts data_table.ID
```

```
##      ID      a      b      c
##  <char> <int> <int> <int>
## 1:      b      1      7     13
## 2:      b      2      8     14
## 3:      b      3      9     15
## 4:      a      4     10     16
## 5:      a      5     11     17
## 6:      c      6     12     18
## [1] "b" "b" "b" "a" "a" "c"
```

```
# subset rows in i
ans = flights[(:origin.eq "JFK") & (:month.eq 6)]
puts ans.head
```

```
# Get the first two rows from flights.
```

```
ans = flights[(1..2)]
puts ans
```

```
# Sort by origin asc, then dest desc (example kept commented):
# ans = flights[E.order(:origin, -(:dest))]
# puts ans.head
```

```
## Empty data.table (0 rows and 11 cols): year,month,day,dep_delay,arr_delay,carrier...
##   year month  day dep_delay arr_delay carrier origin dest air_time
##   <int> <int> <int>    <int>    <int>   <char> <char> <char>    <int>
## 1:  2014     1     1        14        13      AA    JFK    LAX       359
## 2:  2014     1     1         -3        13      AA    JFK    LAX       363
##   distance  hour
##   <int> <int>
## 1:    2475     9
## 2:    2475    11
```

```

# Select column(s) in j
# select arr_delay column, but return it as a vector.

ans = flights[:all, :arr_delay]
puts ans.head

# arr_delay as data.table (not plain vector).

ans = flights[:all, :arr_delay.list]
puts ans.head

ans = flights[:all, E.list(:arr_delay, :dep_delay)]

```

```

## [1] 13 13 9 -26 1 0
##      arr_delay
##      <int>
## 1:      13
## 2:      13
## 3:       9
## 4:     -26
## 5:       1
## 6:       0

```

15 Apache Arrow

Apache Arrow is a **columnar** in-memory format used heavily in R and Python for analytics. In Galaaz, **Ruby does not hold an Arrow C++ table itself**; instead you build ordinary Ruby structures (arrays of row hashes), and `R::Arrow.from_ruby_batches` creates a real **Arrow Table inside GNU R**. From there you use R's `arrow` and `dplyr` packages as usual: `group_by` on the Arrow table, `summarise` for aggregates, then `collect()` to materialize a tibble when you need in-memory R rows.

That pattern matches production use: **JRuby threads** (or sequential code) assemble many rows in Ruby; you pay **one** bridge-heavy handoff to R; `dplyr` runs vectorised work on the Arrow table in R.

Prerequisites: install R packages `arrow` and `dplyr`. Run scripts with `bin/galaaz-jruby` (or the same JVM flags as in `docs/testing.md`) so the Arrow JNI stack is available.

15.1 Other R::Arrow helpers

The Ruby module `R::Arrow` (see `lib/R_interface/r_arrow.rb`) also includes:

- `R::Arrow.table_from(df)` — wrap an R `data.frame` / tibble as an Arrow table.
- `R::Arrow.read_feather` / `write_feather`, `read_parquet`, `dataset(path)` — file and dataset IO on paths visible to R.

15.2 Example: many Ruby rows $\rightarrow$ Arrow in R $\rightarrow$ grouped statistics

The repository test `slow-specs/arrow_large_pipeline_spec.rb` builds 200k rows in parallel (eight threads $\times$ 25,000 rows), pushes them through `R::Arrow.from_ruby_batches`, then checks that `dplyr` group summaries match a Ruby reference calculation. The same logic appears below at a **smaller scale** so this manual can knit quickly; increase `thread_count` and `rows_per_thread` when experimenting locally.

```
# Scaled-down version of slow-specs/arrow_large_pipeline_spec.rb.
unless R::Support.eval("requireNamespace('arrow', quietly=TRUE) && requireNamespace('dplyr'
  puts '(Skip: need arrow + dplyr in R; use bin/galaaz-jruby outside gKnit.)'
else
  thread_count = 4
  rows_per_thread = 500
  group_count = 5

  batches = []
  mutex = Mutex.new
  threads = []

  thread_count.times do |tid|
    threads << Thread.new do
      start = tid * rows_per_thread
      local = (start...(start + rows_per_thread)).map do |i|
        {
          id: i,
          grp: "g#{i % group_count}",
          value: (i % 17) + 1,
          weight: ((i % 5) + 1) * 0.5
        }
      end
      mutex.synchronize { batches << local }
    end
  end
  threads.each(&:join)

  tbl = R::Arrow.from_ruby_batches(batches)
  puts "R class after from_ruby_batches: #{tbl.rclass}"

  grouped = R.dplyr__group_by(tbl, :grp)
  summarised = R.dplyr__summarise(
    grouped,
    n: E.n(),
    total: E.sum(:value),
    wsum: E.sum(:value * :weight)
  )
  out = R.dplyr__collect(summarised)

  puts 'Per-group summary (first rows):'
  puts R.as__data__frame(out).head(10)
```

```

total_n = 0
(1..(out.nrow >> 0)).each { |i| total_n += (out[['n']][i] >> 0) }
puts "Sum of group counts n (should equal #{thread_count * rows_per_thread}): #{total_n}"
end

```

```

## R class after from_ruby_batches: Table
## Per-group summary (first rows):
##   grp   n total   wsum
## 1  g0 400  3589 1794.5
## 2  g1 400  3598 3598.0
## 3  g2 400  3590 5385.0
## 4  g3 400  3599 7198.0
## 5  g4 400  3591 8977.5
## Sum of group counts n (should equal 2000): 2000

```

What to notice: (1) Ruby only sees **Hash** rows and **Ruby Thread** objects; (2) a single **from_ruby_batches** call creates the Arrow table in R; (3) **dplyr__group_by** / **dplyr__summarise** / **dplyr__collect** mirror **dplyr::group_by** / **dplyr::summarise** / **dplyr::collect** on an Arrow-backed table. For a lighter test, see **specs/arrow_from_ruby_batches_spec.rb**; for the full-size benchmark, run **bin/run_slow_rspec slow-specs/arrow_large_pipeline_spec.rb**.

16 Bioconductor and DESeq2

Bioconductor packages are ordinary R packages installed from the Bioconductor repositories. Galaaz does not treat them specially: once installed in **GNU R**, you load them with **R.library** like any CRAN package.

16.1 Installing Bioconductor packages

From an R session (or R **-e** '**...**'), use **BiocManager** (see bioconductor.org):

```

if (!requireNamespace("BiocManager", quietly = TRUE))
  install.packages("BiocManager")
BiocManager::install(c("DESeq2", "airway"))

```

The **airway** package ships the example **SummarizedExperiment** used below. **DESeq2** pulls in several dependencies; the first install can take several minutes.

16.2 Example: DESeq2 on the airway dataset

The script **examples/bioconductor_deseq2_airway/deseq2_airway_galaaz.rb** is the canonical version in the repository. Run it from the **Galaaz repository root** with JRuby, for example:

```
bin/galaaz-jruby examples/bioconductor_deseq2_airway/deseq2_airway_galaaz.rb
```

The workflow in Ruby mirrors a standard DESeq2 vignette:

1. `R.library('DESeq2')` and `R.library('airway')`, then `R.data('airway')` so the object exists in R's global environment.
2. `airway = ~:airway` pulls the experiment into a Galaaz wrapper so you can pass it to R functions as a Ruby value.
3. `R.DESeqDataSet(..., design: (:all.til :cell + :dex))` builds the `DESeqDataSet`. The `(:all.til :cell + :dex)` form is Galaaz's way of passing the one-sided formula `~ cell + dex` (adjust for the design you need).
4. Prefilter rows with almost no counts: `keep = R.rowSums(R.counts(dds)) >= 10` and `dds = dds[keep, :all]`.
5. `dds = R.DESeq(dds)` fits the model; `res = R.results(dds, contrast: R.c('dex', 'trt', 'untrt'))` extracts the treatment contrast (adjust `contrast` for your experiment).
6. Summaries use normal Ruby string interpolation on `R.nrow`, `R.ncol`, `R.colnames`, etc.
7. `R.pdf(...)`; `R.plotMA(res, ...)`; `R.dev__off` writes DESeq2's MA plot (path is relative to the process working directory—use the repo root when running the bundled script).

Related benchmarks and warm-run notes live under `docs/deseq2_airway_benchmark.md` and `examples/bioconductor_deseq2_airway/bench_*.rb`.

Below is the full listing (same as the file in the repository). It is **not** executed while this manual is knitted, because **DESeq2** is heavy and may be absent on the build machine.

```
# Canonical script: examples/bioconductor_deseq2_airway/deseq2_airway_galaaz.rb
# Run: bin/galaaz-jruby examples/.../deseq2_airway_galaaz.rb (repo root).

require 'galaaz'

R.library('DESeq2')
R.library('airway')
R.data('airway')

airway = ~:airway

# Build DESeq2 dataset with one-sided formula: ~ cell + dex.
dds = R.DESeqDataSet(airway, design: (:all.til :cell + :dex))

# Prefilter genes with almost no counts.
keep = R.rowSums(R.counts(dds)) >= 10
dds = dds[keep, :all]

# Fit DE model and extract treatment effect.
dds = R.DESeq(dds)
res = R.results(dds, contrast: R.c('dex', 'trt', 'untrt'))

# Compact sanity outputs for quick verification.
puts "Samples: #{R.ncol(dds)}"
puts "Genes after prefilter: #{R.nrow(dds)}"
puts "Result rows: #{R.nrow(res)}"
puts "Result columns: #{R.colnames(res)}"
puts "Significant genes (padj < 0.05): #{R.sum(res.padj < 0.05, na__rm: true)}"
```

```
res_ordered = res[R.order(res.padj), :all]
puts R.head(R.as__data__frame(res_ordered), 10)

# Standard DESeq2 plot call written to file.
R.pdf('examples/bioconductor_deseq2_airway/plotMA_galaaz.pdf')
R.plotMA(res, ylim: R.c(-5, 5))
R.dev__off
```

If **DESeq2** and **airway** are installed, the next chunk loads the data and prints a short preview (it does **not** run **DESeq** so the manual knits quickly).

```
unless R::Support.eval("requireNamespace('DESeq2', quietly=TRUE) && requireNamespace('airway')") {
  puts '(Skip: install DESeq2 and airway via BiocManager in R to run the full example.)'
} else {
  R.library('DESeq2')
  R.library('airway')
  R.data('airway')
  airway = ~:airway
  puts 'airway object (head of assay / dims via R):'
  puts "ncol(samples): #{R.ncol(airway)}"
  puts R.head(R.assay(airway), 3)
end
```

```
## airway object (head of assay / dims via R):
## ncol(samples): [1] 8
##
##      SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516
## ENSG000000000003      679      448      873      408      1138
## ENSG000000000005       0       0       0       0       0
## ENSG000000000419      467      515      621      365      587
##
##      SRR1039517 SRR1039520 SRR1039521
## ENSG000000000003     1047      770      572
## ENSG000000000005       0       0       0
## ENSG000000000419      799      417      508
```

17 Performance

For realistic analyses, **most wall-clock time is spent inside GNU R** (model fitting, I/O inside R, graphics). The Galaaz **bridge** adds overhead mainly from **starting a session**, **serializing requests**, and **wrapping results** in Ruby objects—not from reimplementing R’s numerical work.

Practical tips:

- Keep **hot loops** in R or vectorized code when possible; use Ruby for orchestration, I/O, and glue.
- **Reuse one process**: running many short scripts cold-starts Ruby, the JVM, and R each time; a long-lived process or repeated calls in one run amortize setup (see benchmarks below).
- **Batch data**: merge shards in Ruby, then call **R::Arrow.from_ruby_batches** (or build one data frame) instead of millions of tiny R calls.

For measured discussion (including DESeq2-style workloads and warm comparisons), see `docs/performance.md` and `docs/deseq2_airway_benchmark.md` in the Galaaz repository.

18 Graphics in Galaaz

Creating graphics in Galaaz is quite easy, as it can use all the power of ggplot2. There are many resources on the web that teach ggplot, so here we give a quick example of ggplot integration with Ruby. We continue to use the `:mtcars` dataset and we will plot a diverging bar plot, showing cars that have ‘above’ or ‘below’ gas consumption. Let’s first prepare the data frame with the necessary data:

```
# :mtcars -> Ruby handle
mtcars = ~:mtcars

# Row labels are not a plot column; copy them to car_name.
mtcars.car_name = R.rownames(:mtcars)

# Z-score mpg (mean/sd on mtcars.mpg); round to 2 decimals.
mtcars.mpg_z = ((mtcars.mpg - mtcars.mpg.mean)/mtcars.mpg.sd).round 2

# ifelse is vectorized: below / above average mpg_z.
mtcars.mpg_type = (mtcars.mpg_z < 0).ifelse("below", "above")

# Sort rows by mpg_z.
mtcars = mtcars[mtcars.mpg_z.order, :all]

# Factor car_name so plot order follows sort.
mtcars.car_name = mtcars.car_name.factor levels: mtcars.car_name

puts mtcars.head
```

```
##              mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Cadillac Fleetwood 10.4   8  472 205 2.93 5.250 17.98  0  0   3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0   3    4
## Camaro Z28         13.3   8  350 245 3.73 3.840 15.41  0  0   3    4
## Duster 360         14.3   8  360 245 3.21 3.570 15.84  0  0   3    4
## Chrysler Imperial  14.7   8  440 230 3.23 5.345 17.42  0  0   3    4
## Maserati Bora       15.0   8  301 335 3.54 3.570 14.60  0  1   5    8
##              car_name mpg_z mpg_type
## Cadillac Fleetwood  Cadillac Fleetwood -1.61  below
## Lincoln Continental Lincoln Continental -1.61  below
## Camaro Z28          Camaro Z28 -1.13  below
## Duster 360          Duster 360 -0.96  below
## Chrysler Imperial   Chrysler Imperial -0.89  below
## Maserati Bora        Maserati Bora -0.84  below
```

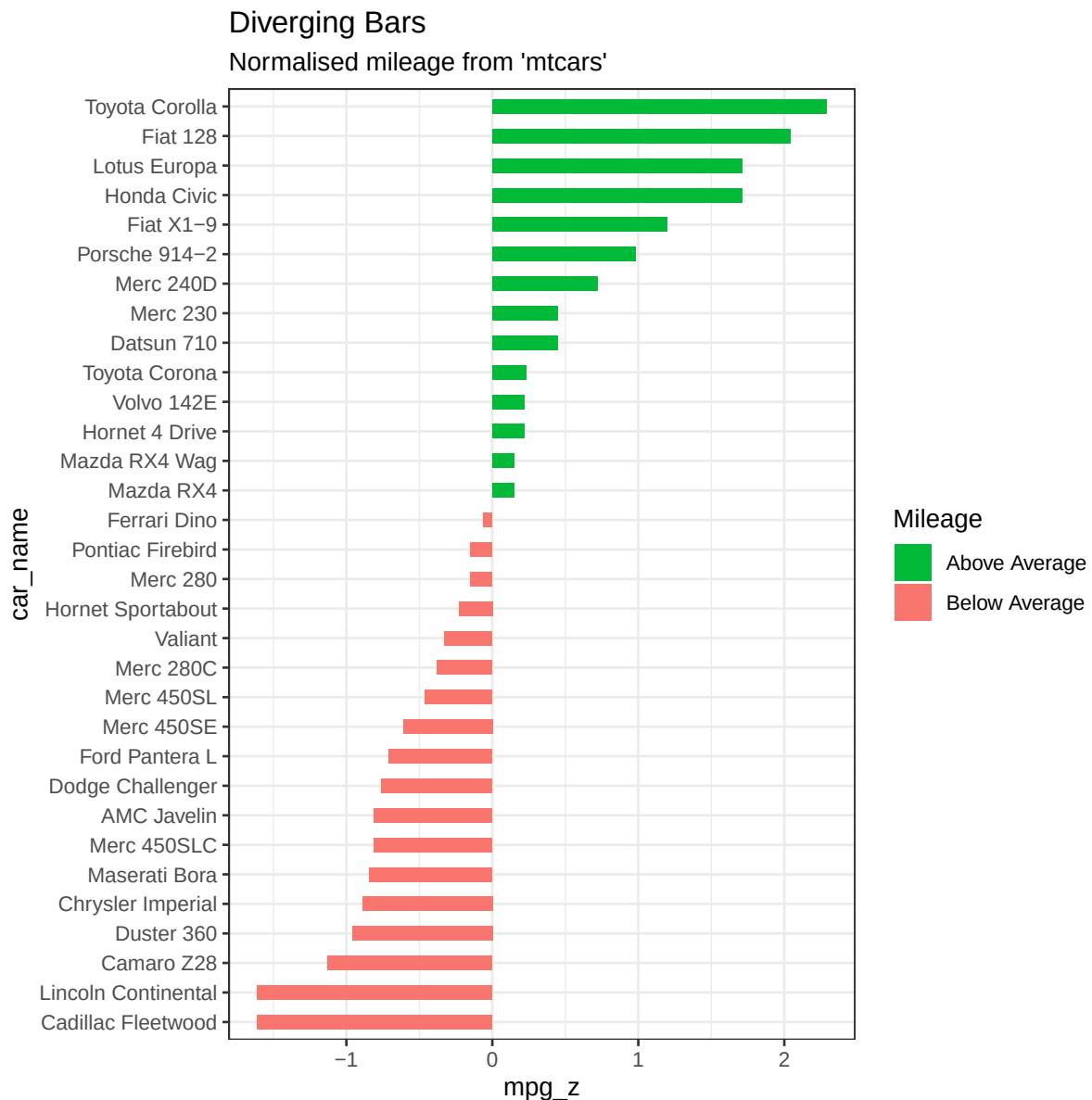
Now, let’s plot the diverging bar plot. When using gKnit, you normally do **not** need to open a graphics device manually; gKnit arranges the figure device for chunk output. Galaaz provides integration with ggplot. The interested reader should check online for more information on

ggplot, since it is outside the scope of this manual describing how ggplot works. Here we give only a brief description of how this plot is generated.

ggplot implements the ‘grammar of graphics’. In this approach, plots are built by adding layers to the plot. On the first layer we describe what we want on the ‘x’ and ‘y’ axis of the plot. In this case, we have ‘car_name’ on the ‘x’ axis and ‘mpg_z’ on the ‘y’ axis. Then the type of graph is specified by adding ‘geom_bar’ (for a bar graph). We specify that our bars should be filled using ‘mpg_type’, which is either ‘above’ or ‘below’ giving then two colours for filling. On the next layer we specify the labels for the graph, then we add the title and subtitle. Finally, in a bar chart usually bars go on the vertical direction, but in this graph we want the bars to be horizontally laid so we add ‘coord_flip’.

```
require 'ggplot'

puts mtcars.ggplot(E.aes(x: :car_name, y: :mpg_z, label: :mpg_z)) +
  R.geom_bar(E.aes(fill: :mpg_type), stat: 'identity', width: 0.5) +
  R.scale_fill_manual(name: 'Mileage',
                      labels: R.c('Above Average', 'Below Average'),
                      values: R.c('above': '#00ba38', 'below': '#f8766d')) +
  R.labs(subtitle: "Normalised mileage from 'mtcars'",
         title: "Diverging Bars") +
  R.coord_flip
```

19 Coding with Tidyverse

In R, and when coding with ‘tidyverse’, arguments to a function are usually not *referentially transparent*. That is, you can’t replace a value with a seemingly equivalent object that you’ve defined elsewhere. To see the problem, let’s first define a data frame:

```
df = R.data__frame(x: (1..3), y: (3..1))
puts df
```

```
##   x y
## 1 1 3
## 2 2 2
## 3 3 1
```

and now, let’s look at this code:

```
my_var <- x
filter(df, my_var == 1)
```

It generates the following error: “object ‘x’ not found.”

However, in Galaaz, arguments are referentially transparent as can be seen by the code below. Note initially that ‘my_var = :x’ will not give the error “object ‘x’ not found” since ‘:x’ is treated as an expression and assigned to my_var. Then when doing (my_var.eq 1), my_var is a variable that resolves to ‘:x’ and it becomes equivalent to (:x.eq 1) which is what we want.

```
my_var = :x
puts df.filter(my_var.eq 1)
```

```
##    x y
## 1 1 3
```

As stated by Hadley

dplyr code is ambiguous. Depending on what variables are defined where, filter(df, x == y) could be equivalent to any of:

```
df[df$x == df$y, ]
df[df$x == y, ]
df[x == df$y, ]
df[x == y, ]
```

In galaaz this ambiguity does not exist, filter(df, x.eq y) is not a valid expression as expressions are build with symbols. In doing filter(df, :x.eq y) we are looking for elements of the ‘x’ column that are equal to a previously defined y variable. Finally in filter(df, :x.eq :y) we are looking for elements in which the ‘x’ column value is equal to the ‘y’ column value. This can be seen in the following two chunks of code:

```
y = 1
x = 2

# looking for values where the 'x' column is equal to the 'y' column
puts df.filter(:x.eq :y)
```

```
##    x y
## 1 2 2
```

```
# looking for values where the 'x' column is equal to the 'y' variable
# in this case, the number 1
puts df.filter(:x.eq y)
```

```
##    x y
## 1 1 3
```

19.1 Writing a function that applies to different data sets

Let's suppose that we want to write a function that receives as the first argument a data frame and as second argument an expression that adds a column to the data frame that is equal to the sum of elements in column 'a' plus 'x'.

Here is the intended behaviour using the 'mutate' function of 'dplyr':

```
mutate(df1, y = a + x)
mutate(df2, y = a + x)
mutate(df3, y = a + x)
mutate(df4, y = a + x)
```

The naive approach to writing an R function to solve this problem is:

```
mutate_y <- function(df) {
  mutate(df, y = a + x)
}
```

Unfortunately, in R, this function can fail silently if one of the variables isn't present in the data frame, but is present in the global environment. We will not go through here how to solve this problem in R.

In Galaaz the method `mutate_y` below will work fine and will never fail silently.

```
def mutate_y(df)
  df.mutate(:y.assign :a + :x)
end
```

Here we create a data frame that has only one column named 'x':

```
df1 = R.data__frame(x: (1..3))
puts df1
```

```
##    x
## 1  1
## 2  2
## 3  3
```

Note that method `mutate_y` will fail independently from the fact that variable 'a' is defined and in the scope of the method. Variable 'a' has no relationship with the symbol ':a' used in the definition of 'mutate_y' above:

```
a = 10
mutate_y(df1)
```

```
## Error in strsplit(path, "/") : non-character argument
## Calls: mutate ... .rlang_purrr_map_mold -> vapply -> FUN -> path_trim_prefix -> strsplit
```

19.2 Different expressions

Let's move to the next problem as presented by Hadley where trying to write a function in R that will receive two arguments, the first a variable and the second an expression is not trivial. Below we create a data frame and we want to write a function that groups data by a variable and summarises it by an expression:

```
library(dplyr)
set.seed(123)

df <- data.frame(
  g1 = c(1, 1, 2, 2, 2),
  g2 = c(1, 2, 1, 2, 1),
  a = sample(5),
  b = sample(5)
)

as.data.frame(df)
```

```
##   g1 g2 a b
## 1  1  1 3 3
## 2  1  2 2 1
## 3  2  1 5 2
## 4  2  2 4 5
## 5  2  1 1 4
```

```
d2 <- df %>%
  group_by(g1) %>%
  summarise(a = mean(a))

as.data.frame(d2)
```

```
##   g1      a
## 1  1 2.500000
## 2  2 3.333333
```

```
d2 <- df %>%
  group_by(g2) %>%
  summarise(a = mean(a))

as.data.frame(d2)
```

```
##   g2 a
## 1  1 3
## 2  2 3
```

As shown by Hadley, one might expect this function to do the trick:

```
my_summarise <- function(df, group_var) {
  df %>%
    group_by(group_var) %>%
    summarise(a = mean(a))
}

# my_summarise(df, g1)
#> Error: Column `group_var` is unknown
```

In order to solve this problem, coding with dplyr requires the introduction of many new concepts and functions such as ‘quo’, ‘quos’, ‘enquo’, ‘enquos’, ‘!’ (bang bang), ‘!!!’ (triple bang). Again, we’ll leave to Hadley the explanation on how to use all those functions.

Now, let’s try to implement the same function in galaaz. The next code block first prints the ‘df’ data frame defined previously in R (to access an R variable from Galaaz, we use the tilde operator ~ applied to the R variable name as a symbol, e.g. :df).

```
puts ~:df
```

```
##   g1 g2 a b
## 1  1  1 3 3
## 2  1  2 2 1
## 3  2  1 5 2
## 4  2  2 4 5
## 5  2  1 1 4
```

We then create the ‘my_summarize’ method and call it passing the R data frame and the group by variable ‘:g1’:

```
def my_summarize(df, group_var)
  df.group_by(group_var).
    summarize(a: :a.mean)
end

puts my_summarize(:df, :g1)
```

```
## # A tibble: 2 x 2
##   g1      a
##   <dbl> <dbl>
## 1     1  2.5
## 2     2  3.33
```

It works!!! Well, let’s make sure this was not just some coincidence

```
puts my_summarize(:df, :g2)
```

```
## # A tibble: 2 x 2
##   g2      a
##   <dbl> <dbl>
## 1     1     3
## 2     2     3
```

Great, everything is fine! No magic, no new functions, no complexities, just normal, standard Ruby code. If you've ever done NSE in R, this certainly feels much safer and easy to implement.

19.3 Different input variables

In the previous section we've managed to get rid of all NSE formulation for a simple example, but does this remain true for more complex examples, or will the Galaaz way prove impractical for more complex code?

In the next example Hadley proposes us to write a function that given an expression such as 'a' or 'a * b', calculates three summaries. What we want a function that does the same as these R statements:

```
summarise(df, mean = mean(a), sum = sum(a), n = n())
#> # A tibble: 1 x 3
#>   mean  sum  n
#>   <dbl> <int> <int>
#> 1     3    15     5

summarise(df, mean = mean(a * b), sum = sum(a * b), n = n())
#> # A tibble: 1 x 3
#>   mean  sum  n
#>   <dbl> <int> <int>
#> 1     9    45     5
```

Let's try it in galaaz:

```
def my_summarise2(df, expr)
  df.summarize(
    mean: E.mean(expr),
    sum: E.sum(expr),
    n: E.n
  )
end

puts my_summarise2(~:df, :a)
puts "
"
puts my_summarise2(~:df, :a * :b)
```

```
##   mean sum n
## 1     3  15 5
##
##   mean sum n
## 1     9  45 5
```

Once again, there is no need to use any special theory or functions. The only point to be careful about is the use of 'E' to build expressions from functions 'mean', 'sum' and 'n'.

19.4 Different input and output variable

Now the next challenge presented by Hadley is to vary the name of the output variables based on the received expression. So, if the input expression is 'a', we want our data frame columns to be named 'mean_a' and 'sum_a'. Now, if the input expression is 'b', columns should be named 'mean_b' and 'sum_b'.

```
mutate(df, mean_a = mean(a), sum_a = sum(a))
#> # A tibble: 5 x 6
#>   g1    g2    a    b mean_a sum_a
#>   <dbl> <dbl> <int> <int>   <dbl> <int>
#> 1     1     1     1     3     3     15
#> 2     1     2     4     2     3     15
#> 3     2     1     2     1     3     15
#> 4     2     2     5     4     3     15
#> # ... with 1 more row
```

```
mutate(df, mean_b = mean(b), sum_b = sum(b))
#> # A tibble: 5 x 6
#>   g1    g2    a    b mean_b sum_b
#>   <dbl> <dbl> <int> <int>   <dbl> <int>
#> 1     1     1     1     3     3     15
#> 2     1     2     4     2     3     15
#> 3     2     1     2     1     3     15
#> 4     2     2     5     4     3     15
#> # ... with 1 more row
```

In order to solve this problem in R, Hadley needs to introduce some more new functions and notations: 'quo_name' and the ':= ' operator from package 'rlang'

Here is our Ruby code:

```
def my_mutate(df, expr)
  mean_name = "mean_#{expr.to_s}"
  sum_name = "sum_#{expr.to_s}"

  df.mutate(mean_name => E.mean(expr),
            sum_name => E.sum(expr))
end

puts my_mutate((~:df), :a)
puts "
"
puts my_mutate((~:df), :b)
```

```
##   g1 g2 a b mean_a sum_a
## 1  1  1 3 3     3    15
## 2  1  2 2 1     3    15
## 3  2  1 5 2     3    15
## 4  2  2 4 5     3    15
## 5  2  1 1 4     3    15
```

```
##
##   g1 g2 a b mean_b sum_b
## 1  1  1 3 3      3    15
## 2  1  2 2 1      3    15
## 3  2  1 5 2      3    15
## 4  2  2 4 5      3    15
## 5  2  1 1 4      3    15
```

It really seems that “Non Standard Evaluation” is actually quite standard in Galaaz! But, you might have noticed a small change in the way the arguments to the mutate method were called. In a previous example we used `df.summarise(mean: E.mean(:a), ...)` where the column name was followed by a `:` colon. In this example, we have `df.mutate(mean_name => E.mean(expr), ...)` and variable `mean_name` is not followed by `:` but by `=>`. This is standard Ruby notation. [explain...]

19.5 Capturing multiple variables

Moving on with new complexities, Hadley proposes us to solve the problem in which the summarise function will receive any number of grouping variables.

This again is quite standard Ruby. In order to receive an undefined number of parameters the parameter is preceded by `*`:

```
def my_summarise3(df, *group_vars)
  df.group_by(*group_vars).
    summarise(a: E.mean(:a))
end

puts my_summarise3(~:df, :g1, :g2)
```

```
## # A tibble: 4 x 3
## # Groups:   g1 [2]
##   g1    g2    a
##   <dbl> <dbl> <dbl>
## 1     1     1     3
## 2     1     2     2
## 3     2     1     3
## 4     2     2     4
```

19.6 Why does R require NSE and Galaaz does not?

NSE introduces a number of new concepts, such as ‘quoting’, ‘quasiquote’, ‘unquoting’ and ‘unquote-splicing’, while in Galaaz none of those concepts are needed. What gives?

R is an extremely flexible language and it has lazy evaluation of parameters. When in R a function is called as `summarise(df, a = b)`, the summarise function receives the literal `‘a = b’` parameter and can work with this as if it were a string. In R, it is not clear what `a` and `b` are, they can be expressions or they can be variables, it is up to the function to decide what `‘a = b’` means.

In Ruby, there is no lazy evaluation of parameters and ‘a’ is always a variable and so is ‘b’. Variables assume their value as soon as they are used, so ‘x = a’ is immediately evaluate and variable ‘x’ will receive the value of variable ‘a’ as soon as the Ruby statement is executed. Ruby also provides the notion of a symbol; ‘:a’ is a symbol and does not evaluate to anything. Galaaz uses Ruby symbols to build expressions that are not bound to anything: ‘:a.eq :b’ is clearly an expression and has no relationship whatsoever with the statment ‘a = b’. By using symbols, variables and expressions all the possible ambiguities that are found in R are eliminated in Galaaz.

The main problem that remains, is that in R, functions are not clearly documented as what type of input they are expecting, they might be expecting regular variables or they might be expecting expressions and the R function will know how to deal with an input of the form ‘a = b’, now for the Ruby developer it might not be immediately clear if it should call the function passing the value ‘true’ if variable ‘a’ is equal to variable ‘b’ or if it should call the function passing the expression ‘:a.eq :b’.

19.7 Advanced dplyr features

In the blog: Programming with dplyr by using dplyr (<https://www.r-bloggers.com/programming-with-dplyr-by-using-dplyr/>) Iñaki Úcar shows surprise that some R users are trying to code in dplyr avoiding the use of NSE. For instance he says:

Take the example of seplyr. It stands for standard evaluation dplyr, and enables us to program over dplyr without having “to bring in (or study) any deep-theory or heavy-weight tools such as rlang/tidyeval”.

For me, there isn’t really any surprise that users are trying to avoid dplyr deep-theory. R users frequently are not programmers and learning to code is already hard business, on top of that, having to learn how to ‘quote’ or ‘enquo’ or ‘quos’ or ‘enquos’ is not necessarily a ‘piece of cake’. So much so, that ‘tidyeval’ has some more advanced functions that instead of using quoted expressions, uses strings as arguments.

In the following examples, we show the use of functions ‘group_by_at’, ‘summarise_at’ and ‘rename_at’ that receive strings as argument. The data frame used in ‘starwars’ that describes features of characters in the Starwars movies:

```
puts (~:starwars).head
```

```
## # A tibble: 6 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex  gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Luke Sky~    172    77 blond      fair        blue         19  male  mascu~
## 2 C-3PO      167    75 <NA>      gold        yellow       112  none  mascu~
## 3 R2-D2       96    32 <NA>      white, bl~  red          33  none  mascu~
## 4 Darth Va~   202   136 none      white       yellow       41.9 male  mascu~
## 5 Leia Org~   150    49 brown     light       brown        19  fema~  femin~
## 6 Owen Lars   178   120 brown, gr~ light       blue         52  male  mascu~
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

The grouped_mean function below will receive a grouping variable and calculate summaries for the value_variables given:

```
library(dplyr)
grouped_mean <- function(data, grouping_variables, value_variables) {
  data %>%
    group_by_at(grouping_variables) %>%
    mutate(count = n()) %>%
    summarise_at(c(value_variables, "count"), mean, na.rm = TRUE) %>%
    rename_at(value_variables, funs(paste0("mean_", .)))
}

gm = starwars %>%
  grouped_mean("eye_color", c("mass", "birth_year"))

## Warning: 'funs()' was deprecated in dplyr 0.8.0.
## i Please use a list of either functions or lambdas:
##
## # Simple named list: list(mean = mean, median = median)
##
## # Auto named with 'tibble::lst()': tibble::lst(mean, median)
##
## # Using lambdas list(~ mean(., trim = .2), ~ median(., na.rm = TRUE))
## Call 'lifecycle::last_lifecycle_warnings()' to see where this warning was
## generated.
```

```
as.data.frame(gm)
```

```
##      eye_color mean_mass mean_birth_year count
## 1      black  76.28571      33.00000      10
## 2       blue  86.51667      67.06923      19
## 3  blue-gray  77.00000      57.00000       1
## 4      brown  66.09231     108.96429      21
## 5      dark      NaN           NaN       1
## 6      gold      NaN           NaN       1
## 7 green, yellow 159.00000           NaN       1
## 8      hazel  66.00000      34.50000       3
## 9      orange 282.33333     231.00000       8
## 10     pink      NaN           NaN       1
## 11     red   81.40000      33.66667       5
## 12  red, blue      NaN           NaN       1
## 13   unknown  31.50000           NaN       3
## 14     white  48.00000           NaN       1
## 15    yellow  81.11111      76.38000      11
```

The same code with Galaz, becomes:

```
def grouped_mean(data, grouping_variables, value_variables)
  data.
  group_by_at(grouping_variables).
  mutate(count = E.n).
  summarise_at(E.c(value_variables, "count"), ~:mean, na_rm = true).
```

```

    rename_at(value_variables, E.funs(E.paste0("mean_", value_variables)))
  end

puts grouped_mean(~:starwars), "eye_color", E.c("mass", "birth_year"))

```

```

## # A tibble: 15 x 4
##   eye_color      mean_mass mean_birth_year count
##   <chr>          <dbl>          <dbl> <dbl>
## 1 black          76.3            33      10
## 2 blue           86.5            67.1    19
## 3 blue-gray      77             57       1
## 4 brown          66.1           109.     21
## 5 dark           NaN             NaN       1
## 6 gold           NaN             NaN       1
## 7 green, yellow  159             NaN       1
## 8 hazel          66             34.5     3
## 9 orange         282.            231      8
## 10 pink          NaN             NaN       1
## 11 red           81.4            33.7     5
## 12 red, blue     NaN             NaN       1
## 13 unknown       31.5            NaN       3
## 14 white         48             NaN       1
## 15 yellow        81.1            76.4    11

```

The examples above cover programmatic dplyr with string column names and `_at` helpers. The same Galaaz patterns (symbols, `E.*` for expression-safe functions, and Ruby methods on R-backed objects) extend to other tidyverse workflows; consult R package documentation for function-specific arguments.

20 Contributing

- Fork it
- Create your feature branch (`git checkout -b my-new-feature`)
- Write tests — use `bin/run_rspec` or `bin/run_all_rspec` with JRuby so JVM flags and the load path match `docs/testing.md`
- Commit your changes (`git commit -am 'Add some feature'`)
- Push to the branch (`git push origin my-new-feature`)
- Open a pull request

References

- Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* (Oxford, UK) 27 (2): 97–111. <https://doi.org/10.1093/comjnl/27.2.97>.
- Wilkinson, Leland. 2005. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag.