**Wait, what is a “skimr”?**

skimr is an R package for summarizing your data. It extends tidyverse packages,  
and dplyr in particular, so that you can get a broad set of summary statistics  
with a single function call.

Library("ropensci/skimr")

skimr is also on  
CRAN, and v2 should  
be appearing there soon. For those of you that might have never seen skimr,  
here’s a typical call.

library(skimr)

library(dplyr)

options(width = 90)

skim(iris)

## ── Data Summary ────────────────────────

## Values

## Name iris

## Number of rows 150

## Number of columns 5

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Column type frequency:

## factor 1

## numeric 4

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Group variables None

##

## ── Variable type: factor ─────────────────────────────────────────────────────────────────

## skim\_variable n\_missing complete\_rate ordered n\_unique top\_counts

## 1 Species 0 1 FALSE 3 set: 50, ver: 50, vir: 50

##

## ── Variable type: numeric ────────────────────────────────────────────────────────────────

## skim\_variable n\_missing complete\_rate mean sd p0 p25 p50 p75 p100 hist

## 1 Sepal.Length 0 1 5.84 0.828 4.3 5.1 5.8 6.4 7.9 ▆▇▇▅▂

## 2 Sepal.Width 0 1 3.06 0.436 2 2.8 3 3.3 4.4 ▁▆▇▂▁

## 3 Petal.Length 0 1 3.76 1.77 1 1.6 4.35 5.1 6.9 ▇▁▆▇▂

## 4 Petal.Width 0 1 1.20 0.762 0.1 0.3 1.3 1.8 2.5 ▇▁▇▅▃

**Getting it right**

Under normal circumstances (i.e. not during a hackathon), most software  
engineering projects begin with a design phase and series of increasingly  
detailed design docs. skimr is only a few hundred lines of code, which means  
“increasingly detailed design docs” translates to one doc. But we did actually  
write it!  
And it still goes a good job of laying out some of the big ideas we were  
interested in taking on for v2.

* Eliminating frictions that resulted from differences in the way we stored  
  data vs how it was displayed to users
* Getting away from using a global environment to configure skimr
* Making it easier for others to extend skimr
* Create more useful ways to use skimr

**Better internal data structures**

In v1, skimr stored all of its data in a “long format”, data frame. Although  
hidden from the user by its print methods, this format would appear any time  
you’d try do something with the results of a skim() call. It looked something  
like this:

skim(mtcars) %>% dplyr::filter(stat=="hist")

# A tibble: 11 x 6

variable type stat level value formatted

1 mpg numeric hist .all NA ▃▇▇▇▃▂▂▂

2 cyl numeric hist .all NA ▆▁▁▃▁▁▁▇

3 disp numeric hist .all NA ▇▆▁▂▅▃▁▂

4 hp numeric hist .all NA ▃▇▃▅▂▃▁▁

5 drat numeric hist .all NA ▃▇▁▅▇▂▁▁

6 wt numeric hist .all NA ▃▃▃▇▆▁▁▂

7 qsec numeric hist .all NA ▃▂▇▆▃▃▁▁

8 vs numeric hist .all NA ▇▁▁▁▁▁▁▆

9 am numeric hist .all NA ▇▁▁▁▁▁▁▆

10 gear numeric hist .all NA ▇▁▁▆▁▁▁▂

11 carb numeric hist .all NA ▆▇▂▇▁▁▁▁

Big ups to anyone who looked at the rendered output and saw that this was how  
you actually filtered the results. Hopefully there are even better applications  
of your near-telepathic abilities.

Now, working with skimr is a bit more sane.

skimmed <- iris %>%

skim() %>%

dplyr::filter([numeric.sd](http://numeric.sd) > 1)

skimmed

## ── Data Summary ────────────────────────

## Values

## Name Piped data

## Number of rows 150

## Number of columns 5

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Column type frequency:

## numeric 1

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Group variables None

##

## ── Variable type: numeric ────────────────────────────────────────────────────────────────

## skim\_variable n\_missing complete\_rate mean sd p0 p25 p50 p75 p100 hist

## 1 Petal.Length 0 1 3.76 1.77 1 1.6 4.35 5.1 6.9 ▇▁▆▇▂

And

dplyr::glimpse(skimmed)

## Observations: 1

## Variables: 15

## $ skim\_type "numeric"

## $ skim\_variable "Petal.Length"

## $ n\_missing 0

## $ complete\_rate 1

## $ factor.ordered NA

## $ factor.n\_unique NA

## $ factor.top\_counts NA

## $ numeric.mean 3.758

## $ [numeric.sd](http://numeric.sd) 1.765298

## $ numeric.p0 1

## $ numeric.p25 1.6

## $ numeric.p50 4.35

## $ numeric.p75 5.1

## $ numeric.p100 6.9

## $ numeric.hist "▇▁▆▇▂"

It’s still not perfect, as you need to rely on a *pseudo-namespace* to refer to  
the column that you want. But this is unfortunately a necessary trade-off.

This normally isn’t something that you have to think too much about, that is  
until you try to combine the means of all your Date columns with the means of  
your numeric columns and everything comes out utterly garbled. So instead of  
that basket of laughs, we prefix columns names by their data type.

There’s a couple of other nuances here:

* The data frame skim() produces always starts off with some metadata  
  columns
* Functions that always produce the same, regardless of input type, can be  
  treated as base\_skimmers and don’t need a namespace

**Manipulating internal data**

A better representation of internal data comes with better tools for reshaping  
the data and getting it for other contexts. A common request in v1 was tooling  
to handle the skimr subtables separately. We now do this with partition(). It  
replaces the v1 function skim\_to\_list().

partition(skimmed)

## $numeric

##

## ── Variable type: numeric ────────────────────────────────────────────────────────────────

## skim\_variable n\_missing complete\_rate mean sd p0 p25 p50 p75 p100 hist

## 1 Petal.Length 0 1 3.76 1.77 1 1.6 4.35 5.1 6.9 ▇▁▆▇▂

You can undo a call to partition() with bind(), which joins the subtables  
into the original skim\_df object and properly accounts for metadata. You can  
skip a step with the function yank(), which calls partition and pulls out a  
particular subtable

yank(skimmed, "numeric")

##

## ── Variable type: numeric ────────────────────────────────────────────────────────────────

## skim\_variable n\_missing complete\_rate mean sd p0 p25 p50 p75 p100 hist

## 1 Petal.Length 0 1 3.76 1.77 1 1.6 4.35 5.1 6.9 ▇▁▆▇▂

Last, with support something close to the older format with the to\_long()  
function. This can be added for something close to backwards compatibility.  
Being realistic on open source sustainability means that we are not able to  
support 100% backward compatibility in v2 even with new functions. Meanwhile you  
can keep using v1 if you are happy with it. However, because skimr’s  
dependencies are under ongoing development, sooner or later skimr v1 will no  
longer work with updates to them.

**Working with dplyr**

Using skimr in a dplyr pipeline was part of the original package design, and  
we’ve needed to devote some extra love to making sure that everything is as  
seamless as possible. Part of this is due to the object produce by skim(),  
which we call skim\_df. It’s a little weird in that it needs both metadata and  
columns in the underlying data frame.

In practice, this means that you can coerce it into a different type through  
normal dplyr operations. Here’s one:

select(skimmed, numeric.mean)

## # A tibble: 1 x 1

## numeric.mean

##

## 1 3.76

To get around this, we’ve added some helper functions and methods. The more  
skimr-like replacement for select() is focus(), which preserves metadata  
columns.

focus(skimmed, numeric.mean)

## ── Data Summary ────────────────────────

## Values

## Name Piped data

## Number of rows 150

## Number of columns 5

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Column type frequency:

## numeric 1

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Group variables None

##

## ── Variable type: numeric ────────────────────────────────────────────────────────────────

## skim\_variable mean

## 1 Petal.Length 3.76

**Configuring and extending skimr**

One big one is customization. We like the skimr defaults, but that doesn’t  
guarantee you will. So what if you want to do something different, we have a  
function factory for that!

my\_skim <- skim\_with(numeric = sfl(iqr = IQR, p25 = NULL, p75 = NULL))

my\_skim(faithful)

## ── Data Summary ────────────────────────

## Values

## Name faithful

## Number of rows 272

## Number of columns 2

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Column type frequency:

## numeric 2

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Group variables None

##

## ── Variable type: numeric ────────────────────────────────────────────────────────────────

## skim\_variable n\_missing complete\_rate mean sd p0 p50 p100 hist iqr

## 1 eruptions 0 1 3.49 1.14 1.6 4 5.1 ▇▂▂▇▇ 2.29

## 2 waiting 0 1 70.9 13.6 43 76 96 ▃▃▂▇▂ 24

Those of you familiar with customizing skim() in v1 will notice a couple  
differences:

* we now has an object called sfl() for managing skimr function lists; more  
  below
* instead of setting global options, we now have a *function factory*

Yes! A function factory. skim\_with() gives us a new function each time we call  
it, and the returned function is configured by the arguments in skim\_with().  
This works the same way as ecdf() in the stats package or colorRamp in  
grDevices. Creating new functions has a few advantages over the previous  
approach.

* you can export a skim() function in a package or create it in a  
  .Rprofile
* you avoid a bunch of potential side effects from setting options with  
  skim\_with()

The other big change is how we now handle different data types. Although many  
will never see it, a key piece of skimr customization comes from the  
get\_skimmers() generic. It’s used to detect different column types in your  
data and set the appropriate summary functions for that type. It’s also designed  
to work with sfl(). Here’s an example from the “Supporting additional objects”  
vignette. Here, we’ll create some skimmers for  
[sf](https://cran.r-project.org/web/packages/sf/index.html) data types:

get\_skimmers.sfc\_POINT <- function(column) {

sfl(

skim\_type = "sfc\_POINT",

n\_unique = n\_unique,

valid = ~ sum(sf::st\_is\_valid(.))

)

}

While it was required in skim\_with(), users must provide a skim\_type value  
when creating new methods. With that, you can export this method in a new  
package (be sure to import the generic), and the new default skimmer is added  
when you load the package.

get\_default\_skimmer\_names()

...

$sfc\_POINT

[1] "missing" "complete" "n" "n\_unique" "valid"

...

Even if you don’t go the full route of supporting a new data type, creating a  
couple of skimr function lists has other benefits. For example, you can add some  
to your .Rprofile as a way to quickly configure skimr interactively.

sfc\_point\_sfl <- sfl(

n\_unique = n\_unique,

valid = ~ sum(sf::st\_is\_valid(.))

)

my\_skimmer <- skim\_with(sfc\_POINT = sfc\_point\_sfl)

**Using skimr in other contexts**

In skimr v1, we developed some slightly hacky approaches to getting nicer  
skim() output in RMarkdown docs. Now, calling skim(), within an RMarkdown doc should produce something nice by default.

skim(chickwts)

|  |  |
| --- | --- |
| **Data summary** | |
| Name | chickwts |
| Number of rows | 71 |
| Number of columns | 2 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Column type frequency: |  |
| factor | 1 |
| numeric | 1 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Group variables | None |

**Variable type: factor**

| **skim\_variable** | **n\_missing** | **complete\_rate** | **ordered** | **n\_unique** | **top\_counts** |
| --- | --- | --- | --- | --- | --- |
| feed | 0 | 1 | FALSE | 6 | soy: 14, cas: 12, lin: 12, sun: 12 |

**Variable type: numeric**

| **skim\_variable** | **n\_missing** | **complete\_rate** | **mean** | **sd** | **p0** | **p25** | **p50** | **p75** | **p100** | **hist** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| weight | 0 | 1 | 261.31 | 78.07 | 108 | 204.5 | 258 | 323.5 | 423 | ▆▆▇▇▃ |

You get a nice html version of both the summary header and the skimr subtables  
for each type of data.

In this context, you configure the output the same way you handle other knitr  
code chunks.

This means that we’re dropping direct support for kable.skim\_df() and  
pander.skim\_df(). But you can still get pretty similar results to these  
functions by using the reshaping functions described above to get subtables. You  
can also still use Pander and other nice rendering packages on an ad hoc basis  
as you would for other data frames or tibbles.

**Wait, that took over a year?**

Well, we think that’s a lot! But to be fair, it wasn’t exactly simple to keep up  
with skimr. Real talk, open source development takes up a lot of time, and the  
skimr developers have additional important priorities. Michael’s family added a  
new baby, and despite swearing up and down otherwise, he got absolutely nothing  
not-baby-related done during his paternity leave (take note new dads!). Elin  
ended up taking a much bigger role on at Lehman, really limiting time for any  
other work.

Even so, these are just the highlights in the normal ebb and flow of this sort  
of work. Since it’s no one’s real job, it might not always be the first focus.  
And that’s OK! We’ve been really lucky to have a group of new users that have  
been very patient with this slow development cycle while still providing really  
good feedback throughout. Thank you all!

We’re really excited about this next step in the skimr journey. We’ve put a huge  
amount of work into this new version. Hopefully it shows. And hopefully it  
inspires some of you to send more feedback and help us find even more ways to  
improve!