Klein-4

Hanwei Hu, Karene Matoka, Chi-Lin Lin

Introduction

In this vignette document, we will demonstrate how to create a S3 class for the Klein_4 group using R package `vctrs`, so make sure you've already installed `vctrs` package before moving forward. S3 is the most commonly used object-oriented programming (OOP) system in R. You can only work on this vignette if you have some knowledge about S3.

Klein four-group(Klein_4) is use in the mathematics and it's a group with four elements. The combination of two elements from the three non-identity elements can produces the the third element, and each element has the identity by combining itself. The Cayley table of Klein four-group is a 4 * 4 matrix with four elements.

This vignette is divided into 2 parts and each part is also divided into different categories.

The first part of this vignette was created to explain the different topics associated with `vctrs` via demonstrations. Those are the topics will discuss on this part:

- How to create a class
- How to build a helper function
- How to add methods and generics
- The concept of coercion and casting
- How to insure equality and comparison between variables
- How to define arithmetic operations

The second part consists of 3 exercises to confirm and check your understanding of the `vctrs` package.

In this vignette document, we will demonstrate how to create a S3 vector for the Klein_4 group using R package `vctrs`, so make sure you've already installed `vctrs` package before moving forward.

```r
# import necessary package

# Check whether the vctrs is installed, if not, install it.
if(!require(vctrs)) {
  install.packages("vctrs")
}
```

```
## Loading required package: vctrs
```

```r
library(vctrs)
```

Part 1

Create Class

An S3 object is any object with a class attribute. For any s3 object, you should have

- A construction function, for internal use only
- A helper function, for users to create objects of our class.

Constructor Fortunately, The vctrs packages provides many useful functions to create construction function for a s3 class.

- `new_vctr()` : create a new vctrs class.
- `vec_assert()` : to checks types and/or sizes

Typically, a constructor should be:

- called `new_myclass` instead of "myclass"
- Having one argument for the base object
- check the type of the base object and types of each argument.

Below is example for how we define a low-level constructor that uses `vec_assert` to check the type of the arguments.

```
# This is an overarching group sgrp
new_sgrp <- function(data = integer(), group = character()) {
  vec_assert(data,integer())
  vec_assert(group, character(), size = 1)
  if(setequal(group,"Klein_4")) {
    data[data > 3L | data < 0L] <- NA_integer_
  }
  new_vctr(data, group = group , class = "sgrp")
}
```

Now we can use this low-level construction for building

```
x <- new_sgrp(c(1L,2L,3L,4L), group = "Klein_4")
x
```

```
## <sgrp[4]>
## [1]  1  2  3 NA
```

```
# check whether it is a s3 object now
sloop::otype(x)
```

```
## [1] "S3"
```

There are a lot of benefits to use `new_vect` to define the construction function compared to using `attr`. The new_vect function could create a class with numerous functions

1. `print()` and `str()` are already defined in terms of `format()`
2. `as.data.frame.vctrs_vctr()` to put your new vector class in a data frame.
3. Subsetting (`[`, `[[`, and `$`), `length<-`, and `rep()` methods are also automatically preserved.
4. Default subsetting assignment methods (`[<-`, `[[<-`, and `$<-`) are also automatically preserved.

```r
example <- new_sgrp(c(1L,2L,3L,4L), group = "sample")

# print and str() function can be used.
str(example)
```

```
##  sgrp [1:4] 1, 2, 3, 4
##  @ group: chr "sample"
```

```r
# is free to use subsetting
print(example[1])
```

```
## <sgrp[1]>
## [1] 1
```

```r
# subsetting assignment also works
example[1] <- 3L
print(example)
```

```
## <sgrp[4]>
## [1] 3 2 3 4
```

```r
# use as.data.frame for free.
as.data.frame(example)
```

```
##   example
## 1       3
## 2       2
## 3       3
## 4       4
```

```r
new_Klein_4 <- function(x = integer()) {
  # check whether the x is integer vector
  vec_assert(x, integer())
  # For each Klein_4 group, the range should be 0:3
  x[x > 3L | x < 0L] <- NA_integer_
  # create it!
  new_sgrp(x, group = "Klein_4")
}
```

```r
new_Klein_4(0:6) -> x
x
```

```
## <sgrp[7]>
## [1]  0  1  2  3 NA NA NA
```

Helper A helper is a function that built to ensure that the users input correctly.

Typically, users are more likely to enter `1` not `1L`. However, our construction is build upon integral, so we need a helper function that can correct users' input.

```
sgrp <- function(x = integer(), group = NA_character_) {
  # make sure the input data is integral.
  # vec_cast can change the x into the type integer
  x <- vec_cast(x, integer())
  group <- vec_cast(group, character())
  new_sgrp(x, group = group)
}

Klein_4 <- function(x = integer()) {
  # make sure the input data is integral.
  # vec_cast can change the x into the type integer
  x <- vec_cast(x, integer())
  new_Klein_4(x)
}
```

Now, even though the input is a double type, the helper function could automatically change the type into the integer and successfully construct the new object.

```
x <- Klein_4(c(0,1,2,3,0,2,3))
x
```

```
## <sgrp[7]>
## [1] 0 1 2 3 0 2 3
```

Check whether or not the function will return a zero-length vector when user doesn't input any argument in it.

```
# It will return a zero-length vector when called with no arguments.
Klein_4()
```

```
## <sgrp[0]>
```

Other Helper (optional)

Test Function It's beneficial to provide a function that tests if an object is of a specific class or type.

```
# check whether the x belongs to Klein_4 class
is_sgrp <- function(x) {
  inherits(x, "sgrp")
}

is_Klein_4 <- function(x) {
  result <- FALSE
  stopifnot(is_sgrp(x))
  if(attributes(x)$group == "Klein_4") {
    return(!result)
  } else return(result)
}
```

```r
x <- Klein_4(c(0,1,2,3))
y <- sgrp(c(1,2,3,4), group = "sample")
is_sgrp(x)
```

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

```r
is_Klein_4(x)
```

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

```r
is_Klein_4(y)
```

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

Now, we could check whether an object belongs to a specific class or type. This would be helpful when I check the input argument in building function.

Extract Functions It is also beneficial to have a function that extracts the attributes of class.

```r
group <- function(x) {
  attr(x,"group")
}
```

```r
x <- Klein_4(0:3)
group(x)
```

```
## [1] "Klein_4"
```

Add Methods and Generics

Format The first function for a class is always the format function. Although the function `new_vect` has already given us a default format function for the class, the function does not return the `group` attribute. Thus, it is best to revise the format function, so we can print the `sgrp` class in a format that we want.

- `formatC()` : This function will output numbers in the vector individually using C style format specifications.

```r
format.sgrp <- function(x, ...) {
  out <- formatC(signif(vec_data(x),3))
  # Makes nicer to print NA instead of "NA"
  out[is.na(x)] <- NA
  cat(attr(x,"group"), "\n")
  out
}
```

```r
# this call for print method
x
```

```
## <sgrp[4]>
## Klein_4
## [1] 0 1 2 3
```

```
# str function
str(x)
```

```
## Klein_4
##  sgrp [1:4] 0, 1, 2, 3
##  @ group: chr "Klein_4"
```

```
# the format method now also output its group.
format(x)
```

```
## Klein_4
```

```
## [1] "0" "1" "2" "3"
```

```
# the orginal method does not return the group attribute
format.default(x)
```

```
## [1] "0" "1" "2" "3"
```

Coercion and Casting

It is always annoy if we do not allow user to change the type of the object, so it is helpful to allow users to change the type if needed.

Coercion

- `coercion`: when the change of the type of variable happens implicitly
- `casting` : when the change of the type of variable happens explicitly

This is to change the prototype of an existing object or to be used when the x and y have different prototypes.

In addition, we can use as functions:

- `vec_ptype2(x, y)`: Use this function when x and y can be safely changed to the same prototype. Otherwise the output will be an error.
- `vec_cast(x, to)` : Use this when you want to change the type of x into another type.

Both generics uses double dispatch. In other words, these generics should be implemented based on the class of two arguments rather than only one.

- `vec_ptype2.sgrp.numeric()`: It would never be called, as the `numeric` is the base type of `double`.
- we could not use `NextMethod` in such double-dispatch generics.

As soon as you've added `vec_ptype2()` and `vec_cast()`, you can use `vec_c()`, `[<-`, and `[[<-`.

```
vec_ptype2.sgrp.sgrp<- function(x, y, ...) {
    new_sgrp(group = group(y))
}
vec_ptype2.sgrp.integer <- function(x, y, ...) y
vec_ptype2.integer.sgrp <- function(x, y, ...) x
```

We can check that we've implemented this correctly with `vec_ptype_show()`:

```
vec_ptype_show(sgrp(), integer(), sgrp())
```

```
## Prototype: <integer>
## 0. (           , <sgrp>    ) = <sgrp>
## 1. ( <sgrp>    , <integer> ) = <integer>
## 2. ( <integer> , <sgrp>    ) = <integer>
```

Casting

Here we define `vec_cast()`:

```
vec_cast.sgrp.sgrp <- function(x,to, ...) {
    new_sgrp(vec_data(x), group = group(to))
}

vec_cast.sgrp.integer <- function(x, to, ...) {
  new_sgrp(x, group = group(to))
}
vec_cast.integer.sgrp <- function(x, to, ...)  vec_data(x)
```

After using `vec_ptype2()` and `vec_cast()`, next step is to use the `vec_c()` implementations, and the function, `[<-`, and `[[<-`.

```
# coercion
vec_c(sgrp(0:4, group = "example"),Klein_4())
```

```
## <sgrp[5]>
## Klein_4
## [1] 0    1    2    3    <NA>
```

```
vec_c(Klein_4(0:3), sgrp())
```

```
## <sgrp[4]>
## NA
## [1] 0 1 2 3
```

```
vec_c(sgrp(0:3, group = "example"), integer())
```

```
## [1] 0 1 2 3
```

```
# This will first convert Klein_4 group into integer and then combine
vec_c(Klein_4(0:3), integer())
```

```
## [1] 0 1 2 3
```

```
vec_c(c(NA,NA,NA,NA),Klein_4())
```

```
## <sgrp[4]>
## Klein_4
## [1] <NA> <NA> <NA> <NA>
```

```
# casting
vec_cast(Klein_4(c(1,2,3,4)), to = integer())
```

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

```
# Notice that double vector cannot cast into a sgrp class.
vec_cast(c(1L,2L,3L,4L), to = sgrp())
```

```
## <sgrp[4]>
## NA
## [1] 1 2 3 4
```

```
x <- Klein_4(0:3)
x
```

```
## <sgrp[4]>
## Klein_4
## [1] 0 1 2 3
```

```
x[1] <- 2L
x[[2]] <- 3L
# Print the x after change
x
```

```
## <sgrp[4]>
## Klein_4
## [1] 2 3 2 3
```

Usually, it is useful to offer conversion functions for our class.

The following examples show how to parse character vectors:

```
as_sgrp <- function(x,...) {
  UseMethod("as_sgrp")
}

as_sgrp.default <- function(x, group = NA_character_,...) {
  vec_cast(x, new_sgrp(group = group))
}
```

```
as_sgrp.character <- function(x, group = NA_character_) {
  value <- as.integer(x)
  new_sgrp(value, group = group)
}
```

```
as_sgrp(c('1','2','3','4'), group = "Klein_4")
```

```
## <sgrp[4]>
## Klein_4
## [1] 1    2    3    <NA>
```

Equality and Compare

As part of building a new S3 vector, you need to implement ways to check for equality and comparison:

There are four proxy generics built within this package to help you with this task:

- `vec_proxy_equal()`: It is going to output a data vector used to compare different components . Others features included are: `==`, `!=`, `unique()`, `anyDuplicated()`, and `is.na()`.
- `vec_proxy_compare()`: The purpose of this proxy is to help users specify how to compare different variables.

Others features included as part of this function are: `<`, `<=`, `>=`, `>`, `min()`, `max()`, `median()`, and `quantile()`.

- `vec_ proxy_order()`: The purpose of this proxy is to help users classify the different components. This proxy can be used with `xtfrm()`, which in turn is called by the `order()`and `sort()` functions.
- `vec_proxy()`: It is going to output a data vector. It is especially helpful when you need field to collect the data. Unless you have to store those data, you do not need to use this proxy frequently.

The goal is to return a simple object such as a data frame or a bare vector. The properties from your class should be also assigned to those objects.

In addition, it is important to specify that: - `vec_proxy_equal` relies on `vec_proxy` - `vec_proxy_compare` relies on `vec_proxy_equal` - `vec_proxy_order` relies on `vec_proxy_compare`

If there is an element in the vector, the new equal function will change the NA into 0.

```
vec_proxy_equal.sgrp <- function(x,...) {
  n <- length(x)
  for(i in 1 : n) {
    if(identical(vec_cast(x[[i]],integer()),NA_integer_)) {
      x[[i]] <- 0L
    }
  }
  vec_cast(x, integer())
}

x <- sgrp(c(0,2,3,NA_integer_))
vec_proxy_equal(x)
```

```
## [1] 0 2 3 0
```

```
x == Klein_4(c(0,2,3,0))
```

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

```
vec_proxy_compare.klein_4 <- function(x,...) {
  n <- length(x)
  for(i in 1 : n) {
    if(identical(vec_cast(x[[i]],integer()),NA_real_)) {
      x[[i]] <- 0L
    }
  }
  vec_cast(x, integer())
  # we are comparing the sum
  sum(vec_cast(x, integer()))
}
```

```
Klein_4(c(1,2,3,NA_real_)) < Klein_4(c(1,2,3,2))
```

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

Arithmetic

The Vctrs package also offers the possibility to perform a broad range of mathematical operations.

- `vec_math(fn, x, ...)` is used to precisely determine the behavior of mathematical functions such as abs(), sum(), and mean().
- `vec_arith(op, x, y)` is used to precisely determine how some arithmetic operations(+, -, and %%) should be behave.
- `vec_math_base` and `vec_arith_base` are also helpful functions. They are mainly used when you need to make use of base R functions.

Each small group has its own table for arithmetic operation. Below table is specific table for the arithmetic operation of Klein_4

```
kl4_cayley <- matrix(c(0L, 1L, 2L, 3L,
                       1L, 0L, 3L, 2L,
                       2L, 3L, 0L, 1L,
                       3L, 2L, 1L, 0L),
                     nrow = 4, ncol = 4)
rownames(kl4_cayley) <- 0:3
colnames(kl4_cayley) <- 0:3
```

Though sum and mean functions make no sense in small group, these functions are important in many classes. We want to make an example for how to design sum and mean functions via vctrs packages.

```
vec_math.sgrp <- function(.fn, .x, ...) {
  switch(.fn,
         vec_math_base(.fn, .x, ...)
  )
}
```

```
x <- Klein_4(0:4)
x
```

```
## <sgrp[5]>
## Klein_4
## [1] 0  1  2  3  NA
```

```
sum(x,na.rm = TRUE)
```

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

```
mean(x,na.rm = TRUE)
```

```
## [1] 1.5
```

```
abs(x)
```

```
## [1]  0  1  2  3 NA
```

`vec_arith()` uses double dispatch. Below is a format of `vec_arith`

```
vec_arith.sgrp <- function(op, x, y, ...) {
  UseMethod("vec_arith.sgrp", y)
}
vec_arith.sgrp.default <- function(op, x, y, ...) {
  stop_incompatible_op(op, x, y)
}
```

We can use `switch()` to check the cases we care about. Then, `stop_incompatible_op()` can throw an informative error message for everything else.

```
vec_arith.sgrp.sgrp <- function(op, x, y, ...) {
  # check whether the group is the via pre-defined group function
  # This group function extract the group attribute of sgrp class.
  stopifnot(group(x) == group(y))
  # create claytable
  table <- kl4_cayley
  return_index <- as.integer(1L + vec_cast(y,integer()) %% 4L)
  x <- table[cbind(as.integer(x) + 1L, return_index)]
  switch(
    op,
    "+" = new_Klein_4(x),
    stop_incompatible_op(op, x, y)
  )
}

vec_arith.numeric.sgrp <- function(op, x, y, ...) {
  # if the x is float, we will discard the decimal, and convert it into the integer.
  x <- as.integer(x)
  # For each group we have special rule of adding a number.
  table <- kl4_cayley
  # initialize the return index as 0 and  we return original if we do nothing
  # the cycle is 4
  return_index <- 1L + as.integer(x) %% 4L
  y <- table[cbind(as.integer(y) + 1L, return_index)]
  switch(
    op,
    "+" = new_Klein_4(y),
    stop_incompatible_op(op, x, y)
    )
}

vec_arith.sgrp.numeric <- function(op, x, y, ...) {
  y <- as.integer(y)
  # For each group we have special rule of adding a number.
  table <- kl4_cayley

  return_index <- 1L + y %% 4L

  x <- table[cbind(as.integer(x) + 1L, return_index)]
  switch(
    op,
    "+" = new_Klein_4(x),
    stop_incompatible_op(op, x, y)
    )
}
```

```
y <- Klein_4(2)
x <- Klein_4(0:3)
x + y
```

```
## <sgrp[4]>
## Klein_4
## [1] 2 3 0 1
```

```
2 + x
```

```
## <sgrp[4]>
## Klein_4
## [1] 2 3 0 1
```

```
x + 0
```

```
## <sgrp[4]>
## Klein_4
## [1] 0 1 2 3
```

```
x + 1
```

```
## <sgrp[4]>
## Klein_4
## [1] 1 0 3 2
```

```r
x + 2
```

```
## <sgrp[4]>
## Klein_4
## [1] 2 3 0 1
```

```r
x + 3
```

```
## <sgrp[4]>
## Klein_4
## [1] 3 2 1 0
```

Part 2

Exercises

1. Create a s3 class called `meter`, which has 2 slots.

- The first slot x is a vector of double type
- The format function should be show his unit: 1 should be shown as 1m
- Another slot is sum, which is the sum of x
- Create a helper function for it
- Create a format function which adds unit when showing the vector

```r
meter(c(1,2,3,4))
```

```
## <meter[4]>
## # Data:  1m 2m 3m 4m
## sum: [1] 10m
```

2. Create a function called `as_meter`, which offers a parsing method for character vectors using `vec_cast`.

```r
my_meter <- as_meter(c('1','2','3','4'))
my_meter
```

```
## <meter[4]>
## # Data:  1m 2m 3m 4m
## sum: [1] 10m
```

3. Create your own `sum` function using `vec_math()` and create your own special `+` function, which plus two meter class together

- The new `sum` function should just return the sum of the meter class, and it should return the unit `m`.
- The new `+` function should also return the unit of data.

```r
m <- meter(c(2,3,4,5))
mean.default(m)
```

```
## [1] 3.5
```

```
mean(m)
```

```
## [1] "3.5m"
```

```
vec_arith.meter <- function(op, x, y, ...) {
  UseMethod("vec_arith.meter", y)
}
vec_arith.meter.default <- function(op, x, y, ...) {
  stop_incompatible_op(op, x, y)
}

vec_arith.meter.meter <- function(op, x, y, ...) {
  switch(
    op,
    "+" = meter(vec_arith_base(op, x, y)),
    "-" = meter(vec_arith_base(op, x, y)),
    stop_incompatible_op(op, x, y)
  )
}
```

```
m <- meter(c(2,3,4,5))
n <- meter(c(1,3,4,5))
m + n
```

```
## <meter[4]>
## # Data:  3m 6m 8m 10m
## sum: [1] 27m
```