ObjectedOrientedProgramming

steppe

2/6/2022

Contents

Assigned Reading: Advanced R chapters 12 - 16

1 Methods

1.1 Libraries

2 Obect Orientated Programming in R

"Object-oriented programming (OOP) is a programming paradigm based on the concept of"objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods)." - from Wikipedia

To date we have largely focused on functional programming. That changes now.

R uses upwards of 25 types of objects. Many of these are highly specialized and we will not go into them, however it is necessary to elaborate on some of the differences between them. To date we have largely been using what may be called 'base' objects, i.e. objects which lack a 'class'. A class contains attributes which define certain parameters of the object, note that each attribute is based off of a 'base' object at some point.

2.1 10.1 Base Objects Reviewed

Some of the most common base objects are vectors, matrices, and lists.

```
rast_vals_char[1:3]
```

```
[1] "Terrestrial" "Terrestrial" "Terrestrial"
```

```
The first element of this vector is: Terrestrial
```

```
A vector is an object of type: base
```

```
This vector objects inhereits of class: character
```

if we use the function 'otype' from the sloop package we can see that this object is a 'base' object and is in a class defined as character

```
raster_matrix <- matrix(c(1,0,1,
                          0,1,1,
                          1,0,0),
                        ncol = 3)
```

1	0	1
0	1	0
1	1	0

```
A matrix is an object of type: base
```

```
A matrix inherits classes of: matrix
A matrix inherits classes of: double
A matrix inherits classes of: numeric
```

```
list
```

```
I made a list out of the last 2 objects and a dataframe:
```

```
$Vector
[1] "Terrestrial" "Terrestrial" "Terrestrial"

$Matrix
     [,1] [,2] [,3]
[1,]    1    0    1
[2,]    0    1    0
[3,]    1    1    0

$Dataframe
  1 2 3
1 1 1 1
2 1 1 1
3 1 1 1
```

```
A list is an object of type: base
```

These base objects are more of less capable of just *storing* values. I.e. In a sense we can put whatever we want into a vector and it will work out - if we have mixed data types theywill be coerced to characters, *but* we can do it.

```
junk <- c(1, 'A', 'alpahbet soup', '$')
print(junk)
```

```
[1] "1"            "A"            "alpahbet soup" "$"
```

2.2 'Introduction' to S3 Objects

```
This is an example of a POSIXct time: 2021-12-05, to me it just looks like a character vector
```

Let's take a look at our vector of POSIXct time zones, which superficially to me (an human) look like numbers separated by dashes.

```
$tzone
[1] "US/Pacific-New"

$class
[1] "POSIXct" "POSIXt"
```

We see that two attributes are stored in that object. The first is the timezone, which is defined elsewhere in R (and I believe is inherited by R from other systems), but is notated within this object. Critically, We also have the formal 'class' definition of a POSIXct format.

```
The amount of seconds since 1970 and 2021-12-05 is: 1638662400
```

POSIXct dates are all actually treated in R as the amount of seconds between an arbitrary date e.g. January 1st 1970, and the date of observation; so R is really keeping dates in a numeric format - but is hiding this from us! And converting the values into a more human friendly format more or less for display purposes only.

This is a new level of functionality that we have not directly considered yet this quarter. As mentioned we have used objects much like book pages. Now we see that certain objects, are able to refer to themselves to perform, for example, calculations and conversions.

```
A POSIXct is an object of type: S3
```

Objects with the capability to store values in fields, and perform procedures upon themselves form the basis of Object Orientated Programming. In R we have two main OOP classes, S3 and S4. In general objects of both of these classes are *'large'*, however this is not always the case.

Another S3 object which superficially looks like a vector is an 'Units' object.

When we define a units object, we can simply supply a number for value, and a unit of measurement.

```
fifty_meters <- units::set_units(50, meter)
print(fifty_meters)
```

```
50 [m]
```

These objects are capable of performing procedures on themselves, such as converting between meters and yards.

```
units(fifty_meters) <- units::make_units(yards)
print(fifty_meters)
```

```
54.68066 [yards]
```

```
A Units is an object of type: S3
```

```
It is its own units class: units
```

```
$units
$numerator
[1] "yards"

$denominator
character(0)

attr(,"class")
[1] "symbolic_units"
```

```
$class
[1] "units"
```

A more shocking S3 object is the data frame

```
S3
```

```
$names
 [1] "j" "b" "e" "i" "h" "f" "d" "a" "g" "c"

$row.names
 [1] "Y" "W" "S" "V" "O" "U" "Z" "N" "T" "R"

$class
[1] "data.frame"
```

We see that of the non-base objects, the most common object is S3. Actually, surprise surprise the amazing data frame has been hiding out as an S3 object this whole time! You may have actually seen an error message indicating this at some point... So if we look at the attributes of a data frame we see that we have both row and columns names, and we also have a formal definition of the class being a data frame, which in part consists of code demanding that our data frame be rectangular in nature.

Finally we turn our attention to one slightly more complex S3 object.

2.2.1 Simple Features

The incredibly popular Simple Features 'SF' package actually stores all it's features in S3 object. Given that a data frame is an S3 object, this is a necessity - but there are some big jumps under the hood. I bet, even more so than these objects being 'accessible' via tidyverse syntax the simplicity of the S3 object compared to S4 sp objects is what spurred there popularity

```
A simple feature is an object of type: S3
```

What is really neat about tibbles, and I am not sure if you all recall is that while they just look like a data frame they are capable of holding a list in a column, hence a 'list column'.

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

For example we can take the four columns of the Iris dataset which contain the measurement variables, and reduce them to two thematic list columns using the 'nest' function.

```
iris <- iris %>%
  nest(petal = starts_with("Petal"),
       sepal = starts_with("Sepal")
       )

iris
```

```
# A tibble: 3 x 3
  Species    petal             sepal
```

```
  <fct>      <list>            <list>
1 setosa     <tibble [50 x 2]> <tibble [50 x 2]>
2 versicolor <tibble [50 x 2]> <tibble [50 x 2]>
3 virginica  <tibble [50 x 2]> <tibble [50 x 2]>
```

```
# Check the first row of the second column
head(iris[[2]][[1]])
```

```
# A tibble: 6 x 2
  Petal.Length Petal.Width
         <dbl>       <dbl>
1          1.4         0.2
2          1.4         0.2
3          1.3         0.2
4          1.5         0.2
5          1.4         0.2
6          1.7         0.4
```

```
# see the structure of the first row of the petal column
str(iris$petal[[1]])
```

```
tibble [50 x 2] (S3: tbl_df/tbl/data.frame)
 $ Petal.Length: num [1:50] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num [1:50] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
```

If we take a close look at the first for of the second column, that for *I. setosa*, we see all of the short and narrow petal measurements we are familiar with for this species.

In this example we see that each element in 'petal' contains a list of its own, each of which has a data frame with a column storing *both* the Petal Length and Petal Width values. Each row of each list column (petal & sepal) we see in this tibble is like this.

So what the developers of SF did is to create an S3 object which can hold all of the spatial information in a list column - without you really realizing this information is there.

```
[1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

```
Warning in as.POSIXlt.POSIXct(x, tz): unknown timezone 'US/Pacific-New'
```

Water	geometry	Date
Pyramid_Lake	POLYGON ((-119.599 40.21368...	2021-12-05
Davis_Lake	POLYGON ((-120.5374 39.9152...	2021-12-05
Frenchman_Lake	POLYGON ((-120.2149 39.936,...	2021-12-05
Silver_Lake	POLYGON ((-119.9159 39.6573...	2021-12-05
Swan_lake	POLYGON ((-119.8356 39.6816...	2021-12-05
Gold_Lake	POLYGON ((-120.6635 39.6694...	2021-12-05

```
$n_empty
[1] 0

$crs
Coordinate Reference System:
  User input: WGS 84
  wkt:
GEOGCRS["WGS 84",
    DATUM["World Geodetic System 1984",
        ELLIPSOID["WGS 84",6378137,298.257223563,
```

```
            LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["latitude",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["longitude",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
    ID["EPSG",4326]]

$class
[1] "sfc_POLYGON" "sfc"

$precision
[1] 50

$bbox
      xmin       ymin       xmax       ymax
-120.66350   39.64317 -119.40150   40.57838
```

This is output is understandably a little much, but these are *all* of the attributes of a Simple Feature Collection which is hidden in the geometry column of a simple feature. Clever right?

2.3 Introduction to S4 Objects

Now S3 objects, are somewhat lax. However there is an S4 object which is a little bit more complex.

Examples of S4 objects include the lovely Raster, and Spatial*(e.g. Points, Polygons, Dataframes...) objects. I find that lists, and S4 objects terrify virtually all of our students each year. Myself included, and in fact I am still slightly spooked by them. While I cannot teach you all how exactly to deal with them, I can teach you all of use them day to day.

S4 objects are not lax, they are the stricter implementation of S3 objects. The classes which compose S4 objects are incredibly well defined, and they are designed for very specific use cases. This may at times make them obnoxious to work with, when you need to build or modify values in them, but thye are worth the pain.

```
writeLines(otype(dec_lakes_sp))
```

```
S4
```

```
writeLines(s3_class(dec_lakes_sp))
```

```
SpatialPolygonsDataFrame
```

2.3.1 Spatial* Objects

Just like as in S3 objects in S4 objects the classes follow certain schema. Note we use a SpatialPolygonsDataFrame as our example here.

In addition to their more strict definitions of classes, S4 objects have another feature which S3 objects lack - Slots. Slots superficially resemble lists in the viewer of RStudio, i.e. they have nested components, but are really their own distinct entities. A SpatialPolygonsDataFrame contains 4 slots, each of these contains a type of data. Note that each slot is accessed using an '@' (still pronounced: 'at') symbol.

```
dec_lakes_sp@plotOrder
```

```
 [1]  1 11  7 13  2  8  3 12 10  5  6  4 14  9
```

```
otype(dec_lakes_sp@plotOrder)
```

```
[1] "base"
```

```
writeLines(s3_class(dec_lakes_sp@plotOrder))
```

```
integer
numeric
```

Here we access the contents of a slot named 'plotOrder'. We see that this slot simply contains an integer vector. What this vector specifies is the order in which the polygons in this object should be plotted if and when used for making maps. While this information is stored simply as an object of the class 'base', other parts of the SpatialPolygonsDataFrame know how to perform operations with this information.

```
head(dec_lakes_sp@data)[,c(1:3,5)]
```

```
Warning in as.POSIXlt.POSIXct(x, tz): unknown timezone 'US/Pacific-New'
```

```
  id          Water Data_source       Date
1  1   Pyramid_Lake   Sentinel2 2021-12-05
2  2     Davis_Lake   Sentinel2 2021-12-05
3  3 Frenchman_Lake   Sentinel2 2021-12-05
4  5    Silver_Lake   Sentinel2 2021-12-05
5  6      Swan_lake   Sentinel2 2021-12-05
6 12      Gold_Lake   Sentinel2 2021-12-05
```

We can see that the 'DataFrame' portion of the SpatialPolygonsDataFrame has actually been relegated to it's own slot as well.

```
attributes(dec_lakes_sp@data)
```

```
$names
[1] "id"          "Water"       "Data_source" "Processing"  "Date"

$class
[1] "data.frame"

$row.names
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

```
otype(dec_lakes_sp@data)
```

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

```
writeLines(s3_class(dec_lakes_sp@data))
```

```
data.frame
```

While the two slots above, @plotOrder & @data, contain relatively simple objects (base and S3 respectively). The remaining two slots, which are both S4 classes contain all of the spatial information.

```
CRS
```

```
[1] "S4"
```

```
$projargs
[1] "+proj=longlat +datum=WGS84 +no_defs"

$class
[1] "CRS"
attr(,"package")
```

```
[1] "sp"

$comment
[1] "GEOGCRS[\"WGS 84\",\n    DATUM[\"World Geodetic System 1984\",\n        ELLIPSOID[\"WGS 84\",63781
```

The first of the S4 slots is the proj4string (Proj is an awesome open source library for performing conversions between projections), which contains a class "CRS" which defines the Coordinate Reference System of this SpatialPolygonsDataFrame. Hence any time you modify the CRS of a SPDF, you are interacting with this slot.

The final, and most complex of the two S4 slots in a SpatialPolygonsDataFrame is the Polygons slot. This is where the coordintaes, and topological information, fo every single polygon is stored.

```
S4

$Polygons
$Polygons[[1]]
An object of class "Polygon"
Slot "labpt":
[1] -120.26621   40.15194

Slot "area":
[1] 0.0002266631

Slot "hole":
[1] FALSE

Slot "ringDir":
[1] 1

Slot "coords":
          [,1]     [,2]
[1,] -120.2684 40.16119
[2,] -120.2577 40.16024
[3,] -120.2604 40.14826
[4,] -120.2657 40.14006
[5,] -120.2657 40.14006
[6,] -120.2758 40.14942
[7,] -120.2684 40.16119



$plotOrder
[1] 1

$labpt
[1] -120.26621   40.15194

$ID
[1] "9"

$area
[1] 0.0002266631

$class
[1] "Polygons"
```

```
attr(,"package")
[1] "sp"

$comment
[1] "0"
```

Here we are looking at the smallest of the polygons in our SPDF (see the plotOrder?). These data are replicated appropriately for each of the other polygons in this object.

While this S3 object is more complex, we can see how the components held in multiple slots are able to work together to perform operations using the data held in disparate fields throughout the object.

2.3.2 Raster Objects

Finally we have the Raster Layer which is a rather large and complex S4 unit, composed of 12 main S4 slots... Each of these then having from 1 to 13 slots (these second slots quite small, and not uncommonly consisting of a single value).

```
Formal class 'RasterLayer' [package "raster"] with 12 slots
  ..@ file    :Formal class '.RasterFile' [package "raster"] with 13 slots
  .. .. ..@ name        : chr ""
  .. .. ..@ datanotation: chr "FLT4S"
  .. .. ..@ byteorder   : chr "little"
  .. .. ..@ nodatavalue : num -Inf
  .. .. ..@ NAchanged   : logi FALSE
  .. .. ..@ nbands      : int 1
  .. .. ..@ bandorder   : chr "BIL"
  .. .. ..@ offset      : int 0
  .. .. ..@ toptobottom : logi TRUE
  .. .. ..@ blockrows   : int 0
  .. .. ..@ blockcols   : int 0
  .. .. ..@ driver      : chr ""
  .. .. ..@ open        : logi FALSE
  ..@ data    :Formal class '.SingleLayerData' [package "raster"] with 13 slots
  .. .. ..@ values    : int [1:13924] 1 1 1 1 1 1 1 1 1 1 ...
  .. .. ..@ offset    : num 0
  .. .. ..@ gain      : num 1
  .. .. ..@ inmemory  : logi TRUE
  .. .. ..@ fromdisk  : logi FALSE
  .. .. ..@ isfactor  : logi FALSE
  .. .. ..@ attributes: list()
  .. .. ..@ haveminmax: logi TRUE
  .. .. ..@ min       : int 0
  .. .. ..@ max       : int 1
  .. .. ..@ band      : int 1
  .. .. ..@ unit      : chr ""
  .. .. ..@ names     : chr ""
  ..@ legend  :Formal class '.RasterLegend' [package "raster"] with 5 slots
  .. .. ..@ type      : chr(0)
  .. .. ..@ values    : logi(0)
  .. .. ..@ color     : logi(0)
  .. .. ..@ names     : logi(0)
  .. .. ..@ colortable: logi(0)
  ..@ title   : chr(0)
  ..@ extent  :Formal class 'Extent' [package "raster"] with 4 slots
  .. .. ..@ xmin: num 697130
```

```
  .. .. ..@ xmax: num 811776
  .. .. ..@ ymin: num 4388466
  .. .. ..@ ymax: num 4502382
  ..@ rotated : logi FALSE
  ..@ rotation:Formal class '.Rotation' [package "raster"] with 2 slots
  .. .. ..@ geotrans: num(0)
  .. .. ..@ transfun:function ()
  ..@ ncols   : int 118
  ..@ nrows   : int 118
  ..@ crs     :Formal class 'CRS' [package "sp"] with 1 slot
  .. .. ..@ projargs: chr "+proj=utm +zone=10 +datum=WGS84 +units=m +no_defs"
  .. .. ..$ comment: chr "PROJCRS[\"unknown\",\n    BASEGEOGCRS[\"unknown\",\n        DATUM[\"World Geo
  ..@ history : list()
  ..@ z       : list()
```

Raster Layers are large enough that we unfortunately cannot go into it much during class, but I do encourage the curious you to investigate it on your own. Do keep in mind that it has a slot of 'CRS' which we just saw in our SP object, and it also has a slot of 'extent' which we defined ourselves earlier.

```
Class: S4
```

```
Class type: RasterLayer
```

```
.SingleLayerData
```

```
integer
numeric
```

If we look at our data slot we see that we have an 'integer numeric' type - which is typically how a raster is loaded. I will concede up to this point I said to think about a raster as matrix, but we see ours is actually loaded with an integer. Remember that in R both matrices and data frames are vectors. Remember a main difference between a vector and a matrix, is that a matrix has dimensions, i.e. it knows how many values need to be in each row. While these data are not explicit here, they can be found in another slot.

3 Object Orientated Programming Bonus: Make our own S3 and s4 objects.

So we approached S3 and S4 objects from a very top down approach there, we looked at those constructed by others. But we can also create our own objects, so we can look at these from the bottom up.

3.1 Create a simple S3 object

Maybe none of you have noticed yet but you will oon enough; your TA has some traits in common with say Luna Lovegood, and finds it hard to make it to meetings and things. Here we makes a course object which he can save into his R environment so he can just click on it to remind him when he is suppose to show up for class.

```
course <- list(
  name = c("Lecture", "Lecture", "Laboratory", "Laboratory"),
  wing = c('L', 'L','M','L'),
  room = c(170, 170, 166, 62),
  day  = c('T', 'TH', 'F', 'F'),
  time = c('3:30-4:50', '3:30-4:50', '12:00-12:50', '2:00-3:50')
)

writeLines(sloop::s3_class(course)) # we have only made a list so far
```

list

```
writeLines(sloop::otype(course))
```

base

```
class(course) <- "Class_times" # by setting a class attribute we have
# created an S3 object

writeLines(sloop::s3_class(course))
```

Class_times

```
sloop::otype(course)
```

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

```
rm(course)
```

We see that the above object is honestly, just a list that we arbitrarily made an S3 object. That is more or less what it takes to become an S3 object, us just saying hey 'this is a class' !

But we can do things with S3 objects which make them useful to construct. For example, we can add quality assurance checks to our objects.

3.2 Create a more complex S3 object

Your TA made a very poor first list of places he had to be for class, and ended up wandering around tech lost for an hour. He decided to make a slightly more robust Sw object so this did not happen again.

```
course <- function(n, w, r, d, t){

  values <- list(name = n,
                 wing = w,
                 room = r,
                 day = d,
                 time = t)

  '%notin%' <- Negate('%in%')
  type <- c('Lecture', 'Laboratory', 'Seminar')
  tech_letters <- LETTERS[1:13]
  days <- c('M', 'T', 'W', 'TH', 'F', 'S', 'SU')

  if(any(w %notin% tech_letters)) stop("This wing is not valid")
  if(any(d %notin% days)) stop("This day is not valid")
  if(any(n %notin% type)) stop("This course type is not valid")

  attr(values, "class") <- "Course"
  return(values)

}
```

```
course_success <- course(c("Lecture", "Lecture", "Laboratory", "Laboratory"),
                         c('L', 'L','M','L'),
                         c(170,170,166,62),
                         c('T','TH','F','F'),
                         c('3:30-4:50','3:30-4:50', '12:00-12:50', '2:00-3:50')
                         )
```

Here we see that an object of the S3 class can be much more than merely a collection of attributes. An S3 object can perform quality assurance steps to ensure the data comply to certain types, and that values are in appropriate units etc.

If the data are these thoroughly defined, we then see this opens the opportunity for an S3 object to act upon other parts of itself. While in the example above we put in values which match the acceptable criteria of the class we have defined, in the example below will violate the standards of our S3 Object.

```
course_fail <- course("Lecture",
            'Z', # THIS IS NOT DEFINED
            213, 'TH', '2:00-3:50')
```

simulated output (R does not like errors, intentional or not):

"Error in course("Lecture", "Z", 213, "TH", "2:00-3:50") : This wing is not valid"

In the above example, if you input the wrong Wing in Tech, or the wrong day of week abbreviation, this class will angrily let you know and refuse to take your input. I assume this error can be relegated to a warning - but we will not get into those aspects of coding in this class.

One, possible, draw back of an s3 object is that it is relatively lax.

3.3 Create a simple S4 object

The s4 object is not lax. Here we define the data type which each of these slots will accept. If the value you try to put in does not match, the object will not be created.

```
setClass("Office_Hours",
         slots=
           list(
            Instructor="character",
            Wing= "character",
            Number="numeric",
            Days="character",
            Time="numeric",
            Smartroom="logical",
            Windows="logical"
            )
         )
```

- Seven slot object, each slot with a specified data type.
- Will only input to each column of the correct data type.

```
s4_ob_office_hours <- new("Office_Hours",
  Instructor = c('Benkendorf','Scholl','Benkendorf','Scholl'),
  Wing = c('F', 'F', 'G', 'B'),
  Number = c(380, 380, 278, 138),
  Days = c("M", "W", 'T', 'F'),
  Time = c(8, 11, 9, 3),
  Smartroom = c(TRUE, F, T, F),
  Windows = c(FALSE, T, T, F)
)
```

In it's simplest form an s4 object may be constructed via the setClass function. While this object is able to regulate the data types which are entered to each column, it cannot do much more.

3.4 Create a more complex S4 object

An S4 object can have validity functions, which basically ensures the data you put into it is appropriate.

```
office_hrs <- setClass("Office_Hours",

         slots=c(
           Instructor="character",
           Wing= "character",
           Number="numeric",
           Days="character",
           Time="numeric",
           Smartroom="logical",
           Windows="logical"
           ),

        validity=function(object){

            '%notin%' <- Negate('%in%')
            tech_letters <- LETTERS[14:26]
            days <- c('M', 'T', 'W', 'TH', 'F', 'S', 'SU')
            instructors <- c('Scholl', 'Benkendorf')

            if(any(object@Instructor != instructors))stop("This Instructor is not valid")
            if(any(object@Wing %in% tech_letters))stop("This Wing is not valid")
            if(any(object@Days %notin% days)) stop("This Day is not valid")
  }
)
```

- Mandates the appropriate data type is entered
- Checks that the 'Wing' we enter exists in the Tech Building
- Ensures that a valid Instructor is entered
- Ensures that the appropriate abbreviation for a day is entered.
- Object clearly capable of performing procedures on itself.

```
s4_ob_office_hours <- office_hrs(
  Instructor = c('Scholl', 'Benkendorf', 'Scholl', 'Benkendorf'),
  Wing = c('A', 'F', 'B', 'B'),
  Number = c(123, 412, 278, 138),
  Days = c("M", "W", 'T', 'F'),
  Time = c(8, 11, 9, 3),
  Smartroom = c(TRUE, F, T, F),
  Windows = c(FALSE, T, T, F)
)

otype(s4_ob_office_hours)
```

```
[1] "S4"
```

```
s3_class(s4_ob_office_hours)
```

```
[1] "Office_Hours"
attr(,"package")
[1] ".GlobalEnv"
```

```
rm(s4_ob_office_hours, office_hrs)
```

- Easy examples, but we could also script in conversion functions like with the ealier Units and POSIX S3 objects.
- S3/S4 objects can have a lot going on 'under the hood'

- At their heart, they are performing operations on themselves.

You will realize in short time, that the biggest hurdle to dealing with spatial data in R is how complex some of the structures may be. But I guarantee you all have seen more of the intricate workings of these objects than the vast majority of folks which utilize them. Just remember, to appease the validity functions and you will be fine.

4 Works Cited

Advanced R. Wickham, H. https://adv-r.hadley.nz/index.html Accessed 01.09.2022

https://geocompr.robinlovelace.net/spatial-class.html Accessed 01.09.2022

Hijman, R. 05.12.2019 'The raster Package'

https://www.datamentor.io/r-programming/s3-class/ Accessed 01.18.2022

https://rspatial.org/raster/RasterPackage.pdf Accessed 01.09.2022

Pebesma, E.J., R.S. Bivand, 2005. Classes and methods for spatial data in R. R News 5 (2).

Pebesma, E. https://r-spatial.github.io/sf/articles/sf1.html Accessed 01.10.2022

https://cran.r-project.org/web/packages/vctrs/vignettes/s3-vector.html Accessed 01.14.2022

https://en.wikipedia.org/wiki/Object-oriented_programming Accessed 01.19.2022