JSON and MongoDB in R

May 2019

PhillyR x Philadelphia MongoDB User Group - May 2019

"the most ambitious crossover event in history"

Many thanks to our sponsors

Introduction to R

mongoDB.

Language Features

- Turing complete
- High-level
- Functional (at its heart)
- 1-indexed
- Everything is an object

For more technical and rigorous introduction to R language, read Hadley Wickham's (new) Advanced R https://adv-r.hadley.nz/

Variables

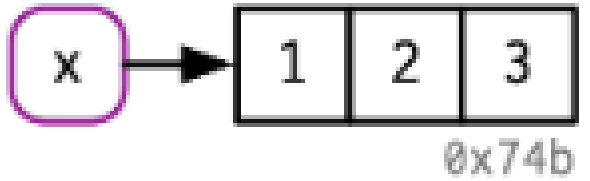
- R has the typical data types you normally find in other languages
  - Boolean
    - T, F, TRUE, FALSE
  - Integer
    - 1L, 1.2e1L, 0xDEADL
  - Double (floating point)
    - 3.14, 1.23e1, 0xDEADBEEF
  - Character
    - “a”, ‘b’, “c”, ‘d”
  - Complex
    - 1+2i
  - Raw (for binary data)
    - 00 12 34

```
> myInteger <- 1L
> myInteger
[1] 1
> myDouble <- 3.14; myDouble
[1] 3.14
> (myCharacter <- "Hello")
[1] "Hello"
> myComplex <- 1 + 2i
> myComplex
[1] 1+2i
```

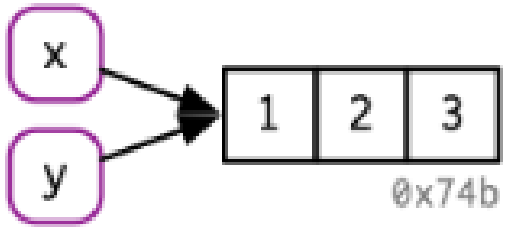
Variables

- Variable assignment by reference

```
> x <- c(1, 2, 3)
```



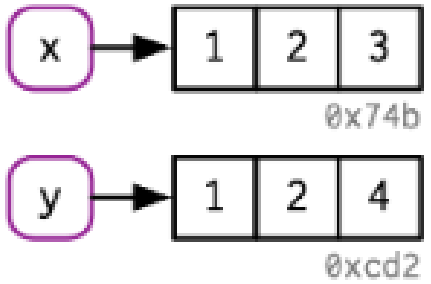

```
> x <- c(1, 2, 3)
> y <- x
```



Variables

- Copy-on-modify (aka immutability)
- aka “R is slow”

```
> y[[3]] <- 4
```



Data structure - vectors

- x and y are both *vectors*.
- Think of vector as being composted of *scalar* of same type (integer, double, boolean, or character)
  ≈ array of primitive in other languages
- Scalar is a vector of length 1
- A vector of length 1 ≈ primitive in Python
- i.e. [1] ≈ 1
  (in R, `c(1) == 1`)

** Very over-simplified and crude (and incorrect) explanations / comparisons in order to prime you for the upcoming slides on R ←→ JSON. Things are a lot more subtle. If you love computer science concepts and want to learn more, seriously take a look at Advanced R book

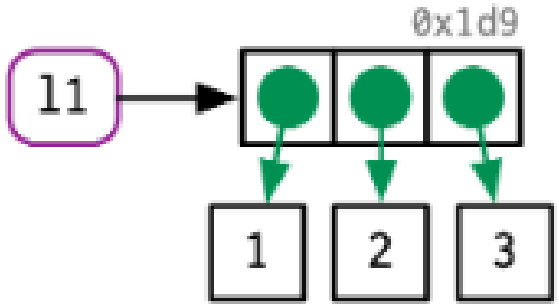
Data structure - lists

- Just like how variable name points to values, elements of a vector can point to values, but in this case it would be a *list*
- ≈ array of variables ?! **

```
> (l1 <- list(1, 2, 3))
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
```



** These are just approximation / alternative explanation. RTARB (Read The Advanced R Book)!

Data structure - lists

- This allows you to have heterogenous values (different types) for each element of a list
- “variable name” concept applies here
- Note that here we use an equal sign instead of an arrow

```
> (l2 <- list(1L, "Hi", 3.14))
[[1]]
[1] 1

[[2]]
[1] "Hi"

[[3]]
[1] 3.14

> (l3 <- list(int = 1L, char = "Hi", double = 3.14))
$int
[1] 1

$char
[1] "Hi"

$double
[1] 3.14
```

Data structure - lists

- An element in a list can be anything, even another list.

```
> (l4 <- list(
+     outerElem1 =
+         list(innerElem1 = c(1,2,3),
+              innerElem2 = LETTERS[1:5]),
+     outerElem2 = "This is complicated but flexible"
+     )
+ )
$outerElem1
$outerElem1$innerElem1
[1] 1 2 3

$outerElem1$innerElem2
[1] "A" "B" "C" "D" "E"


$outerElem2
[1] "This is complicated but flexible"
```

Data structure - lists

- An element in a list can be anything, even another list.

```
> (l4 <- list(
+     outerElem1 =
+         list(innerElem1 = c(1,2,3),
+              innerElem2 = LETTERS[1:5]),
+     outerElem2 = "This is complicated but flexible"
+     )
+ )
$outerElem1
$outerElem1$innerElem1
[1] 1 2 3

$outerElem1$innerElem2
[1] "A" "B" "C" "D" "E"


$outerElem2
[1] "This is complicated but flexible"
```

(inner) List with two elements, each with vector of different size and data type

Nested elements in list are easily accessible by indexing sequentially

Data structure - lists

- An element in a list can be anything, even another list.

```
> (l4 <- list(
+     outerElem1 =
+         list(innerElem1 = c(1,2,3),
+              innerElem2 = LETTERS[1:5]),
+     outerElem2 = "This is complicated but flexible"
+     )
+ )
$outerElem1
$outerElem1$innerElem1
[1] 1 2 3

$outerElem1$innerElem2
[1] "A" "B" "C" "D" "E"


$outerElem2
[1] "This is complicated but flexible"
```


Vector of length 1

Data structure - lists

- Alternative way of looking at this complex structure

{

“outerElem1” : … ,

“outerElem2” : “This is complicated but flexible”

}

```
> (l4 <- list(
+     outerElem1 =
+         list(innerElem1 = c(1,2,3),
+              innerElem2 = LETTERS[1:5]),
+     outerElem2 = "This is complicated but flexible"
+     )
+ )
$outerElem1
$outerElem1$innerElem1
[1] 1 2 3

$outerElem1$innerElem2
[1] "A" "B" "C" "D" "E"


$outerElem2
[1] "This is complicated but flexible"
```

Data structure - lists

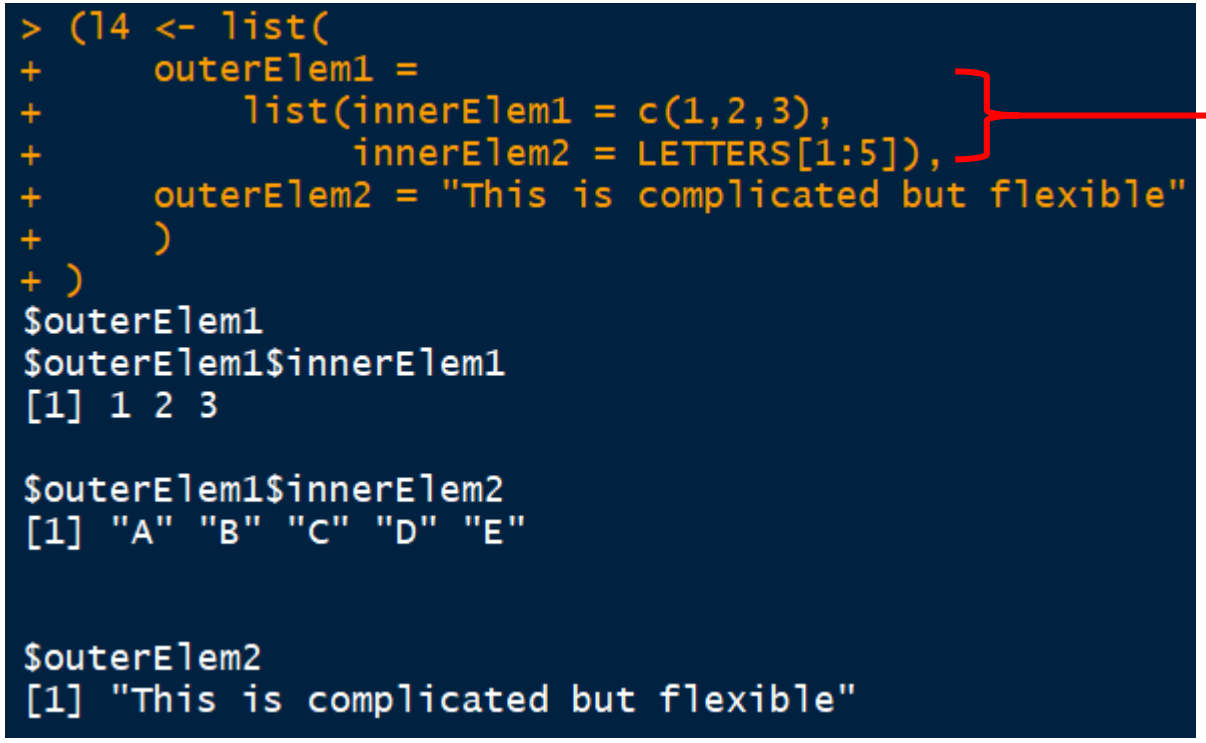
- Alternative way of looking at this complex structure

{

“outerElem1” : … ,

“outerElem2” : “This is complicated but flexible”

}



```
> (l4 <- list(
+     outerElem1 =
+         list(innerElem1 = c(1,2,3),
+              innerElem2 = LETTERS[1:5]),
+     outerElem2 = "This is complicated but flexible"
+     )
+ )
$outerElem1
$outerElem1$innerElem1
[1] 1 2 3

$outerElem1$innerElem2
[1] "A" "B" "C" "D" "E"


$outerElem2
[1] "This is complicated but flexible"
```

Data structure - lists

- Alternative way of looking at this complex structure

{
“outerElem1” : … ,
“outerElem2” : “This is complicated but flexible”
}

```
> (l4 <- list(
+     outerElem1 =
+         list(innerElem1 = c(1,2,3),
+              innerElem2 = LETTERS[1:5]),
+     outerElem2 = "This is complicated but flexible"
+     )
+ )
$outerElem1
$outerElem1$innerElem1
[1] 1 2 3

$outerElem1$innerElem2
[1] "A" "B" "C" "D" "E"


$outerElem2
[1] "This is complicated but flexible"
```

{
“innerElem1” : [1, 2, 3],
“innerElem2” : [“A”, “B”, “C”, “D”, “E”]
}

Data structure - lists

- Alternative way of looking at this complex structure

```
{
    “outerElem1” : … ,
    “outerElem2” : “This is complicated but flexible”
}
```

```
> (l4 <- list(
+     outerElem1 =
+         list(innerElem1 = c(1,2,3),
+              innerElem2 = LETTERS[1:5]),
+     outerElem2 = "This is complicated but flexible"
+     )
+ )
$outerElem1
$outerElem1$innerElem1
[1] 1 2 3

$outerElem1$innerElem2
[1] "A" "B" "C" "D" "E"


$outerElem2
[1] "This is complicated but flexible"
```

```
{
    “innerElem1” : [1, 2, 3],
    “innerElem2” : [“A”, “B”, “C”, “D”, “E”]
}
```

We shall call this curly bracket-y format – JSON!

R - JSON “rules”

- Named lists become JSON object

```
> (l4 <- list(
+     outerElem1 =
+         list(innerElem1 = c(1,2,3),
+              innerElem2 = LETTERS[1:5]),
+     outerElem2 = "This is complicated but flexible"
+     )
+ )
```

```
{
    “outerElem1” : … ,
    “outerElem2” : …
}
```

- Unnamed list becomes JSON array of array elements

```
> (l5 <- list(LETTERS[1:3], 1:3))
[[1]]
[1] "A" "B" "C"

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

```
[
    [ … ],
    [ … ]
]
```

- Anything that is / can be named → { “name” : <<value>> }

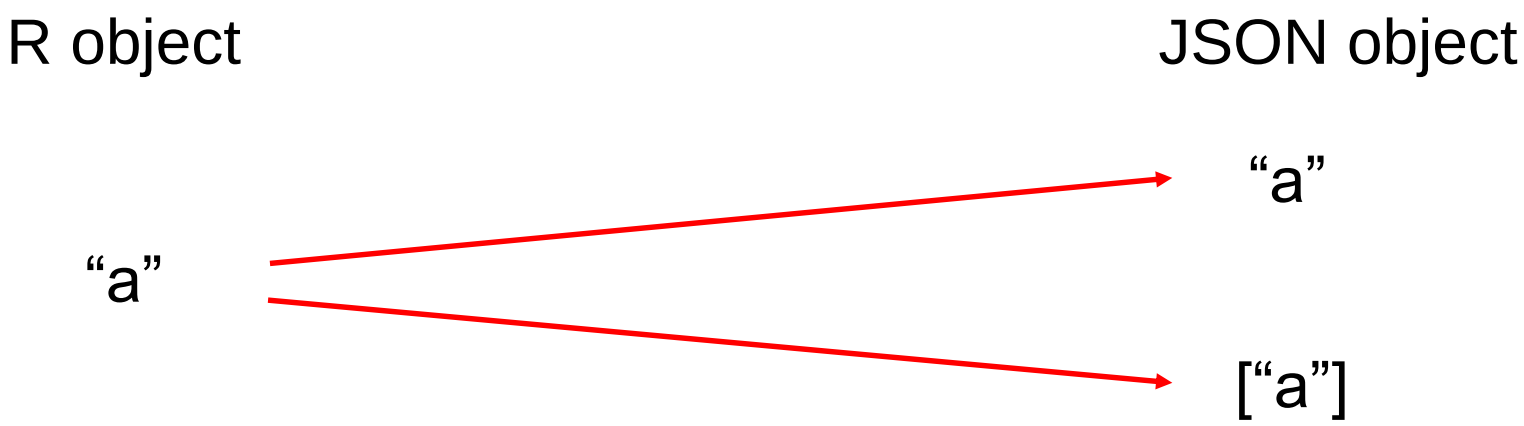
R - JSON “rules”

- R data types are intuitively converted

Booleans	
T, F, TRUE, FALSE	[true, false, true, false]
Integers	
1L, 1.2e1L, 0xDEADL	[1, 12, 57005]
Double (floating point)	
3.14, 1.23e1, 0xDEADBEEF	[3.14, 12.3, 3735928559]
Character	
“a”, ‘b’, “c”, “d”	[“a”, ‘b’, “c”, “d”]
Complex	
1+2i	??
Raw (for binary data)	
00 12 34	??

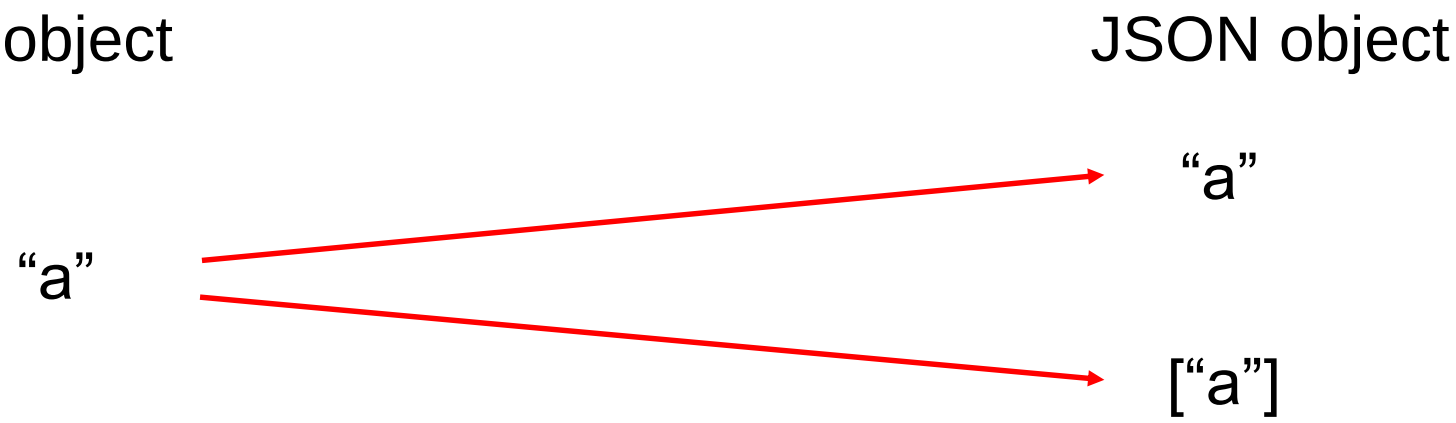
R - JSON problems

- Should R vector of length 1 be a JSON array?

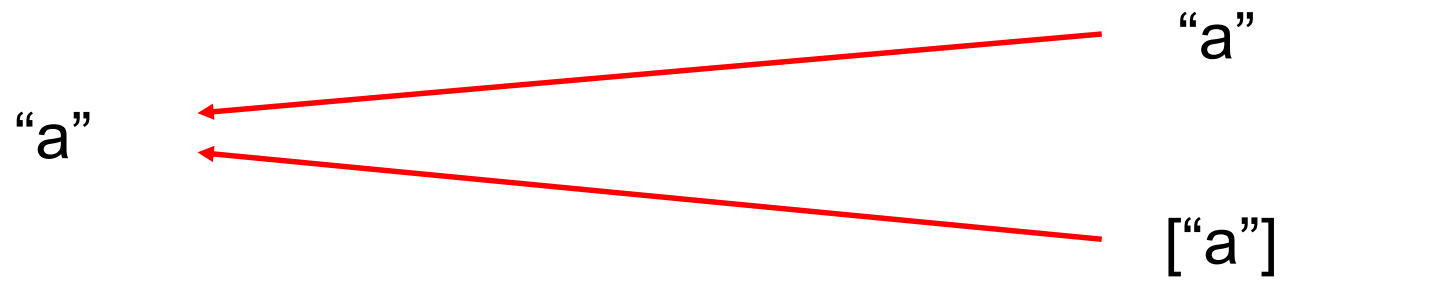


R - JSON problems

- Should R vector of length 1 be a JSON array?



- JSON to R conversion is more troubling!



R - JSON problems

- In R, there are `NA` and `NULL` values for different types of missingness. How would this represent this in JSON?
- Conversely, how do you represent JSON `null` into R object?
- How do you represent more complex R objects, like `complex`, `raw`, `factor`, `Date`, and `POSIXt`?
- How do you represent higher dimension R objects, like `matrix` and `data.frame`?
- How do you represent other metadata associated with complex R objects, like factor levels, row names for `data.frame`?

R - JSON conversion using `library(jsonlite)`

- `toJSON(...)` to convert R object into JSON

- `fromJSON(...)` to convert JSON (represented as R's character) into R object

- Automatic conversion of complex R objects with consistent default rule settings. These can be overwritten if neccessary
  - R vectors are always converted to JSON array.
  - Complex R & JSON objects are mapped in R-user friendly way

- See vignette https://cran.r-project.org/web/packages/jsonlite/vignettes/json-aaquickstart.html

- `library(rjson)` also exists, but `library(jsonlite)` is more widely used today due to more consistent rule and better maintenance.

Why use R and JSON?

- JSON is widely used for language / technology agnostic data transfer format
- Use `library(httr), library(opencpu), library(plumber)` to query HTTP API that returns results as JSON or productionize R code as HTTP API
- NoSQL databases often use JSON-like data format for transferring data between DB server and your R session.
- Using MongoDB database is facilitated by `library(jsonlite)` and `library(mongolite)`.

Demo

```
library(mongolite)
library(tidyverse)

# Free book! https://jeroen.github.io/mongolite/

# look for sample collection "listingsAndReviews" on "sample_airbnb"
m <- mongo(
  db = "sample_airbnb",
  collection = "listingsAndReviews",
  url = "mongodb+srv://phillyr:risawesome@phillyr-djozr.azure.mongodb.net/test?retryWrites=true",
  verbose = T
)

# How many documents? i.e. SELECT COUNT(*) FROM listingsAndReviews
m$count('{}')

# Query only one, i.e. SELECT * FROM listingsAndReviews LIMIT 1
oneTrueListing <- m$find(fields = '{}', limit = 1)

# Is automatically a data.frame
class(oneTrueListing)
colnames(oneTrueListing)

# tibblify to view data easily
(oneTrueListing <- tibble::as_tibble(oneTrueListing))

# Using iterate to get 1 value as JSON (by passing automatic conversion to dataframe)
findOne_asJSON <- m$iterate()
oneTrueListing_json <- findOne_asJSON$json(1)
# Print as pretty
jsonlite::prettify(oneTrueListing_json)
```

```
# let's remove summary, space, description, neighborhood_overview, and notes because they really long texts
jsonlite::prettify(
  m$iterate(
    query = sprintf('{ "_id": "%s" }', oneTrueListing$`_id`),
    fields = '{"summary" : false, "space" : false, "description" : false, "neighborhood_overview" : false, "notes" : false }',
    limit = 1)$json(1)
)


# Some of the fields are "complex". Let's explore
simpleListing <- m$find(
  query = sprintf('{ "_id": "%s" }', oneTrueListing$`_id`),
  fields = '{"summary" : false, "space" : false, "description" : false, "neighborhood_overview" : false, "notes" : false }'
)

# What is the class of each column in data.frame?
sapply(simpleListing, function(x) {paste(class(x), collapse = "/")})

# Which column is not a vector?
colnames(simpleListing)[!sapply(simpleListing, is.vector)]
```

```
# Example of nested document
jsonlite::prettify(
  m$iterate(
    query = sprintf('{ "_id": "%s" }', oneTrueListing$`_id`),
    fields = '{"_id" : true, "beds" : true, "price": true, "images" : true }',
    limit = 1)$json(1)
)

# Watch what happens to "price" and "images"
(nestedObjects <- m$find(
  query = sprintf('{ "_id": "%s" }', oneTrueListing$`_id`),
  fields = '{"_id" : true, "beds" : true, "price": true, "images" : true }'
))

class(nestedObjects$images)
nestedObjects$images

# flattens non-recursively, leading to 4-col tibble with "images" column being a data.frame
as_tibble(nestedObjects)
sapply(as_tibble(nestedObjects), function(x) {paste(class(x), collapse = "/")})
```

```
# What if the value was an array? (e.g. "amenities")
class(simpleListing$amenities)
(nestedArray <- m$find(
  query = sprintf('{ "_id": "%s" }', oneTrueListing$`_id`),
  fields = '{"_id" : true, "beds" : true, "price": true, "images" : true, "amenities" : true }'
))

class(nestedArray$amenities)
nestedArray$amenities

# flattens non-recursively, leading to 5-col tibble with "images" column being a data.frame,
# and "amenties" as a list
as_tibble(nestedArray)
sapply(as_tibble(nestedArray), function(x) {paste(class(x), collapse = "/")})
```