Week 3 Walkthrough: Lists

James Robertson

2025-01-17

Big 3 Object Types

Coming to you ~~live~~ asynchronously from my couch, it's **ST 308 Content Walkthroughs**! As always, I'm your host, James Robertson.

We will be continuing this week's walkthrough with a discussion of the flexible **Lists**! Before we get into that, let's remind ourselves of the relevant vocabulary.

- **Collection**: A group of things that we're considering all together. The objects inside my grocery cart, for example, form a **collection**. The students in this class also form a **collection**! The **object types** we're looking at today are all special types of **collections**!
- **Elements**: The *things* inside of a **Collection** are **Elements**. My **grocery collection** always has eggs and milk in it, so eggs is an **Element** and so is milk.
- **Homogeneous**: Every **element** into the same category. Going to the grocery store and only buying milk, cheese, yogurt, and ice cream would lead to me having a pretty **homogeneous** diet as everything is in the same category (dairy in this case).
- **Heterogeneous**: **Elements** can fall into different categories! Getting meat, dairy, fruits, and vegetables at the grocery store is what the surgeon general and food pyramid recommend, so a **heterogeneous** (varied) diet is generally considered ideal. A new (tastier) strain of brussels sprouts entered the mainstream in the last decade so eating leafy greens is only getting easier!

Whew, still a lot! Now that we're all set to get back into the exciting stuff, let's meet the marvelous **List**!

Lists

Name:	List
Related to:	Data Frame
Likes:	Order, names, double square brackets, nesting, growth, yoga
Dislikes:	Math
Bio:	Born from `list()` in a crossfire hurricane with a lifegoal of bringing order to chaos.

- **Lists** are **ordered collections** of **heterogeneous elements**.
  - *More jargon!* **Lists** are our most flexible object type, letting you store *anything you want* in an ordered way so there still is always a first, always a last.

But I need you to understand that when I say you can store *anything* in a **list**, I mean *anything*. Text? Numbers? Easy! **Vectors**? Now we're getting somewhere! More **lists**? You're beginning to believe.

The price of this flexibility is mathematics. We can't do *any* mathematical operations on **lists** as a whole (even **lists** of numbers!), but we can grab individual **elements** to do things with. Another great thing about **lists** is how easy it is to reference **elements** by **name**! Let's start by making a big, messy list with **elements** *numEl*, *charEl*, and *listEl*.

```
aList<-list(numEl=c(1, 3, 9, 27),
            charEl=c('Words','words','words'),
            listEl=list(larry='Larry',
                        layer1=list(layer2=list(layer3='We have to go deeper'))))
```

For effect I've layered the **list element** to have a *ton* of layers of **lists** in **lists**, but sometimes you might actually want that! I think of **lists** of **lists** as akin to file trees, letting you navigate to what you want rather than having a separate object. For reasons. I promise there are good ones, but that's a topic I will once again relegate to office hours!

Now that we have our **list**, most of what we do with it is **indexing** to reach specific **elements** and operate on them like normal. This is *very* similar to **vector** indexing, but we will use *double* square brackets [[ ]]. Let's see some examples.

```
print(aList[[1]] + 5)
```

```
## [1]  6  8 14 32
```

```
print(aList[['numEl']] + 5)
```

```
## [1]  6  8 14 32
```

```
print(aList[[2]][1])
```

```
## [1] "Words"
```

```
print(aList[[3]][[2]][[1]][[1]])
```

```
## [1] "We have to go deeper"
```

Referencing by numerical **index** like this obviously works, but the real convenience comes in from the **names** we assigned. Using `$` we can reference an **element** by **name** instead of having to either (a) count to its index or (b) remember its numerical index! More examples!

```
print(aList$numEl)
```

```
## [1]  1  3  9 27
```

```
print(aList$charEl)
```

```
## [1] "Words" "words" "words"
```

```
print(aList$listEl$layer1$layer2$layer3)
```

```
## [1] "We have to go deeper"
```

And what's *really* convenient is getting to reference them by abbreviated names! After the `$` you can type *just enough* to uniquely identify the **element** (i.e. no other name starts that way) and save *a lot of typing.*

```
print(aList$n)
```

```
## [1]  1  3  9 27
```

```
print(aList$c)
```

```
## [1] "Words" "words" "words"
```

```
print(aList$l$l$l$l)
```

```
## NULL
```

And we've run into an error! When going deep into our list, *something* went wrong. Like I mentioned before, you have to specify enough to *uniquely* identify the **element**. In this case, inside the **element** `listEl` there's

an **element** named `larry` and one named `layer1`! I *want* `layer1`, but `larry`'s in the list too and if all I tell `R` is "`l`", then it can't know which I mean. Since they both start with "`la`" I also can't just do that, I have to go on to at least specify "`lay`", like below.

```
print(aList$l$lay$l$l)
```

```
## [1] "We have to go deeper"
```

So when indexing a **list** and getting **null**, probably something in your indexing is misspecified, but it could be a problem with your **list**. If you reference an **element** that doesn't actually exist, **lists** will always still output `NULL` instead of an error!

```
print(aList$ImAgInAtIoN)
```

```
## NULL
```

This idiosyncratic behavior actually is part of a really useful behavior! Adding new **elements** to **vectors** is a pain and you are basically *remaking* the **vector** every time. Not so with **lists**! If I want to add a new element, I can just use the **assignment arrow** `<-` and overwrite this `NULL` value with what I want!

```
aList$ImAgInAtIoN <- "A boatload of fancy sushi"
print(aList$ImAgInAtIoN)
```

```
## [1] "A boatload of fancy sushi"
```

And voila! My heart's deepest desire has overwritten what once was a `NULL` value. This handy behavior is *exactly* why I often create **lists** of **lists**! It can get unwieldy if you're not careful, but if you rely on auto-complete you can almost eliminate having to remember variable names at all! You just then have to give your **list elements** *descriptive names* or you'll be back to square one.

Lists *want* to be your friend! They are *incredibly* flexible and useful, but can be kind of intimidating with their capacity for complexity. If you take your time, though, the **list** will become a constant ally.