Highlights of R

The *Why*, but not the *How*, of Using R

Samuel Lurie

Southern California Linux Expo 16x (2018)

({Sunday, March 11, 2018}$'1500-1600')[[Room 104]]

Overview

R has made expensive proprietary statistical software irrelevant.

- R launched in 1993, Python in 1991 (both FOSS)
- Before then, you had SAS (first release in 1972)
- SAS is still around, but it's $9720 and not needed any more. (https://www.sas.com/store/products-solutions/cSoftware-p1.html?storeCode=SAS_US)

Inertia

Why SAS?

Why do so many organizations use SAS? It's clunky, difficult to read, and feels so archiac compared to other languages like R and Python.

Because we use SAS and *that's* the way things are done.

Inertia. Dinosaurs at my firm who haven't learned a new thing in decades will stage a ... revolt if we stop paying an arm-and-a-leg for that shouty, verbose, idiotic language.

–excerpts of a thread on the statistics subreddit (tinyurl.com/whysas)

SAS tries to convince us of its continued relevance...

...but just digs itself into a deeper hole.

I think [R] addresses a niche market ... but [w]e [at SAS] have customers who build engines for aircraft. I am happy they are not using freeware when I get on a jet.
—Anne H. Milley (SAS), 2009

(http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html?pagewanted=all)

What about Python?

But we already have Python.

So why should we care about R?

In terms of functionality...

R and Python have been imitating each other for years.

- They regularly port each other's libraries. (e.g. ggplot2 and Pandas)
- Jupyter notebooks now have support for R.
- As data science languages, both are full-featured and mature.
- What you can do in one, you can generally do in the other. The one glaring exception: web development capability

The question can no longer be answered by appealing to functionality!

So why choose R over Python? (or vice versa)

- design philosophy
  - Python:
    - performance > (freedom and flexibility)
  - R:
    - (freedom && flexibility) > performance
- stylistic differences and nuances (later in this talk)
- Base R is far more comprehensive than base Python.
- At the end of the day:
  - "For 95% of programming problems, the best language is the one that you're best at."

    —Andrew Robinson (https://www.youtube.com/watch?v=ZIUcI_OYbd8, 55min)

In R, operators are functions.

Trivial Examples

- `+`(2,3) $\mapsto$ 5 (i.e. same as 2+3)
- `/`(4,12) $\mapsto$ .3333 (i.e. same as 4/12)

Parentheses are a function.

- A pair of parentheses is a call to the identity map function!
- This is why using more parentheses slows down R code.

Braces are a function.

- A pair of braces is a call to a function that returns the last variable calculated.
- This is why functions (delineated by braces) do not need an explicit return statement.

A Black Sheep among Languages

- R indexes from 1. (more on this soon)
- usage among non-programmers and "monolingual" R users
- R has four different assignment operators. (more on this soon)
- assignment into function calls??
  - e.g. `c("Length", "Width") -> names(df1)`

Indexing from 1

Old habits die hard.

- surprisingly controversial
- There are benefits to this approach. As an example, take x = 1:5

Negative Indices

```
> print(x)
1 2 3 4 5
> x = x[-3]
> print(x)
1 2 4 5
```

Indexing with Size Functions

```
> x[length(x)]
5
```

Four Different Assignment Operators

the operators, in order of decreasing precedence:

- `assign('x', 1.645)` # optional envir arg for specific namespace
- `1.645 -> x` (rightward assignment, also `->>`)
- `x <- 1.645` (leftward assignment, also `<<-`)
- `x = 1.645`

a couple amusing effects of R's order of operations:

- `a = b <- 3`
- `a = ((b <- 3) + 1)` # Remember that operators are functions!

But R is Slow!

- R has a reputation for slow execution.
- This is not so much a fault of R, but rather the price paid for implementing R's philosophy.
- The real problem lies in one big misconception about R execution times—loops are slow!

List Comprehension (part 1 of 2)

The apply family of functions are syntactic sugar for applying a function over all/selected elements of a variable.

- They are apply, lapply, sapply, mapply, and tapply.
- Each has its own usage.
  - sapply(1:100, function(x){2*sqrt(x)+1}) returns a numeric vector of $2\sqrt{1}+1$, $2\sqrt{2}+1$, ... $2\sqrt{100}+1$
  - `lapply(list(   c(1,2),c(3,4)    ),sum)`

    ```
    [[1]]
    [1] 3
    [[2]]
    [1] 7
    ```
  - tapply: a wrapper around lapply, used for blocking
  - mapply: multiple vectors inputted into a function

List Comprehension (part 2 of 2)

Recycling

Functions defined for one element can take multiple elements anyway.

example: factorial()

```
factorial(1:6)
1        2        6        24       120      720
```

example: length mismatches

```
(1:6)/c(10,100,1000)
0.1 0.02 0.003 0.4 0.05 0.006
```

apply() vs. loops

- The R community largely views loops as being slow and advocates using the apply functions as a faster alternative.
  - This is misinformation! What slows down loops is when you have memory allocations in each iteration!
    - apply may indeed be faster—but that's simply because it acts as a guard against programmers doing that.

apply() vs. loops: A Counterexample

loop

```
# .4 seconds
proc.time() -> exec_time
vec_a <- 1:1e6
vec_b <- vector(mode = "numeric", length = 0)
for(i in vec_a)
{
    vec_b[[1 + length(vec_b)]] <- sqrt(i)
    # notwithstanding a memory operation in each iteration!
}
exec_time <- proc.time() - exec_time
print(exec_time)
```

sapply()

```
# .7 seconds
proc.time() -> exec_time
vec_a <- 1:1e6
vec_b <- sapply(vec_a, sqrt)
exec_time <- proc.time() - exec_time
print(exec_time)
```

Parallelization

- An upside of the apply() habit:
  - R programmers have been conditioning themselves to write computations in embarrassingly parallel ways. Even before parallel computing was in vogue!
- Libraries for parallel computing (CPU) thus came to R very naturally as parallelized wrappers for the apply() functions.
  - e.g. The parallel package has the mclapply() function, which performs an lapply() in parallel.
    - It has an argument for specifying the number of cores to use. This argument can even be set as detectCores()-1!
- Similarly, R has libraries for GPU computing.

Landmarks in R beyond present scope

- I would be remiss if I did not mention these in an R talk, even though they are beyond our present scope.
  - ggplot2
  - shiny

But R isn't Deployable!

- R has long been criticized and belittled for its apparent lack of deployability.
  - Until a few years ago, it was hard to refute this criticism.
  - But now... we have Docker!

my R::Shiny Dockerfile


```
FROM openanalytics/r-base

# "forked" from https://www.shinyproxy.io/deploying-apps/

RUN apt-get update && apt-get install -y \
    sudo \
    pandoc \
    pandoc-citeproc \
    libcurl4-gnutls-dev \
    libcairo2-dev \
    libxt-dev \
    libssl-dev \
    libssh2-1-dev \
    libssl1.0.0

RUN R -e "install.packages(  c( \"shiny\",\"rmarkdown\",\"tibble\",\"formattable\",
\"dplyr\",\"stringi\",\"knitr\",\"rvest\",\"XML\",\"pbapply\",\"ggplot2\",\"chron\",
\"lubridate\",\"DistributionUtils\",\"stringr\",\"reshape2\",\"grDevices\",
\"outliers\")  , repos = \"http://cran.cnr.berkeley.edu/\", dependencies=TRUE)"

COPY app.R /etc

CMD ["R", "-e", "shiny::runApp('/etc/app.R', port = 8080, host = '0.0.0.0')"]
```

Being Resourceful

- But R still doesn't deploy well! You can't compile a stand-alone "desktop" R script!
    - Yes, you can. If you're resourceful. (I'll tell you one way to do it.)
    - Let that be the takeaway. The hacker spirit of R is why I thought this talk is suitable for a GNU/Linux conference.
        - I hope you think so too.

The End

$\}$