ETC4500/ETC5450 Advanced R programming

Week 12: Interfacing with other languages

SETUs

SETUs are now open.

Please complete your SETU, we make improvements based on your feedback.

This is especially important for us in this unit, since it is our first year running the unit!

https://monash.edu/ups/setu

R as an interface language

R is a powerful design language, with lots of flexibility for creating good (or bad) programming interfaces.

Much of R is built up on libraries from other languages, and R's flexible interface design makes them easy to use.

R Core

R itself is mostly written using different programming languages (mostly C and Fortran).

You can find the source code for R at https://svn.r-project.org/R/, or mirrored on GitHub at https://github.com/wch/r-source

Wrapper functions and abstractions

The use of abstraction and wrapping other software is fundamental to programming.

Wrapper functions call a second function with minimal/no change to the output. They are used to adapt existing code to work for a new design or programming language.

Wrappers often involve abstraction, a process of reducing complexity by simplifying the user-facing function’s design.

Wrapping functions with NSE

Last week we saw how non-standard evaluation (NSE) can take any syntactically valid R code and evaluate it differently.

Metaprogramming is often used to directly translate R code into other languages.

Interfacing other programming languages

An interface to a different programming language involves:

- Designing an R interface which can be translated into code for the other language
- Converting objects to and from each language
- Passing side-effects (like image output)

Interfacing Python with reticulate

```r
library(reticulate)
```

The `reticulate` package allows Python to run from within R.

- Translates R syntax to Python
- Converts R objects to Python
- Converts Python objects to R

The Python version and package environment can be set with:

```r
use_python("/usr/local/bin/python")
```

Python example from R with reticulate

```r
# reticulate::py_install("numpy")
np <- import("numpy", convert = FALSE)

# do some array manipulations with NumPy
a <- np$array(c(1:4))
a
```

```
[1, 2, 3, 4]
```

```r
sum <- a$cumsum()
sum
```

```
[1, 3, 6, 10]
```

```r
# convert to R explicitly at the end
py_to_r(sum)
```

```
[1]  1  3  6 10
```

Converting objects between R and Python

Full conversion table here: *Calling Python*

```r
r_to_py(1)
```

```
1.0
```

```r
r_to_py(1:10)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```r
r_to_py(list(norm = rnorm(10), pois = rpois(10, 3)))
```

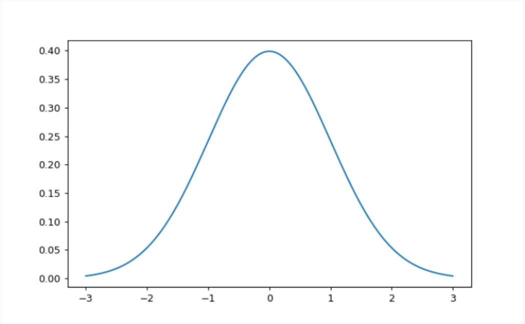
```
{'norm': [0.39930144926135686, -0.25397988300359914, 0.03104403494514269, -1.188400
```

```r
r_to_py(mtcars)
```

```
Dict (11 items)
```

Plots (and other side-effects)

```r
plt <- import("matplotlib.pyplot")
fig <- plt$figure(figsize=c(14,8))
x <- seq(-3, 3, by = 0.01)
plt$plot(x,dnorm(x))
plt$show()
```



Interfacing other programming languages

- Any system commands with `system()`
- C/C++: Directly in R with `.Call()` or Rcpp (next week!)
- Julia: JuliaCall
- Matlab/Octave: R.matlab
- Stata: RStata
- JavaScript: V8
- Java: rJava
- Lua: luajr

Data analysis with databases

Often data for analysis is stored and used within a database.

A database is an efficient way of securely storing and interacting with large datasets.

It is also a good technique for working with data that is too large to fit in memory.

dbplyr

```r
library(dbplyr)
```

The dbplyr package allows you to use dplyr code to manipulate tables from databases.

It achieves this using non-standard evaluation to convert dplyr and R code into suitable database code for a connected database.

dbplyr backends

Backends are interfaces between R and database languages.

There are many database backends available for dbplyr:

- MySQL / SQLite
- Snowflake
- PostgreSQL
- Spark
- ODBC
- MS Access
- SAP HANA
- Hive
- Impala
- Oracle
- Redshift
- Teradata

Creating a database

You can quickly create a SQLite database in memory with:

```
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
con
```

```
<SQLiteConnection>
  Path: :memory:
  Extensions: TRUE
```

Currently this database doesn’t contain any tables:

```
DBI::dbListTables(con)
```

```
character(0)
```

Using a database

We can add a dataset to the database from R with:

```r
copy_to(con, mtcars)
DBI::dbListTables(con)
```

```
[1] "mtcars"       "sqlite_stat1" "sqlite_stat4"
```

You can then retrieve the table using `tbl()`

```r
tbl(con, "mtcars")
```

```
# Source:   table<`mtcars`> [?? x 11]
# Database: sqlite 3.51.1 [:memory:]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
 2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
```

Manipulating a table

With the database table object, you can use dplyr:

```r
tbl(con, "mtcars") |>
  group_by(cyl) |>
  summarise(mean(mpg), mean(hp))
```

```
# Source:   SQL [?? x 3]
# Database: sqlite 3.51.1 [:memory:]
    cyl `mean(mpg)` `mean(hp)`
  <dbl>       <dbl>      <dbl>
1     4        26.7       82.6
2     6        19.7      122. 
3     8        15.1      209. 
```

Collecting the results

When your dplyr data manipulation is complete, you can then `collect()` your results as a regular R data frame for use with other packages.

```
tbl(con, "mtcars") |>
  group_by(cyl) |>
  summarise(mean(mpg), mean(hp)) |>
  collect()
```

```
# A tibble: 3 x 3
    cyl `mean(mpg)` `mean(hp)`
  <dbl>       <dbl>      <dbl>
1     4        26.7       82.6
2     6        19.7      122.
3     8        15.1      209.
```

Disconnecting from a database

Once finished, it is good practice to disconnect from the database using `DBI::dbDisconnect()`:

```r
DBI::dbDisconnect(con)
```

See now that the database is disconnected:

```r
con
```

```
<SQLiteConnection>
  DISCONNECTED
```

Connecting to a remote database

In most cases you will be connecting to a remove database.

Here's the credentials to a PostgreSQL database containing some very important data:

- Host: arp.nectric.com.au:5432
- Username: monash
- Password: arp2024
- Database: arp

Connecting to a remote database

Your turn!

Connect to the remote database and use the data.

Hint: the connection code looks like this:

```r
con <- DBI::dbConnect(
  RPostgres::Postgres(),
  dbname = "???",
  host = "???", port = "???",
  user = "???", password = "???"
)
```

Using data on a remote database

As before, dbplyr allows you to manipulate tables using dplyr code.

```
tbl(con, "penguins") |>
  group_by(species) |>
  summarise(avg_mass_g = mean(body_mass_g, na.rm = TRUE))
```

All database operations are done on the remote server.

How it works - translating expressions

dbplyr uses NSE to translate R code into SQL / database code.

You can try this out directly with `translate_sql()`:

```r
translate_sql(mean(body_mass_g, na.rm = TRUE), con = con)
translate_sql(x ^ 2L, con = con)
translate_sql(substr(x, 5, 10), con = con)
```

How it works - translating expressions

Not all R functions can be translated to database queries.

Consider `logp1()`, it gets translated directly as SQL:

```r
translate_sql(logp1(x), con = con)
```

However this doesn't work, `log(body_mass_g + 1)` does.

```r
tbl(con, "penguins") |>
  mutate(logp1(body_mass_g))
```

How it works - translating expressions

Not all database queries can be written in R.

For this you can write literal `SQL` commands with the `sql()` function.

```
translate_sql(sql("x!"), con = con)
translate_sql(x == sql("ANY VALUES(1, 2, 3)"), con = con)
```

How it works - translating dplyr verbs

For any chain of dplyr commands, you can find the SQL / database query by using `show_query()` instead of `collect()`.

```r
tbl(con, "penguins") |>
  group_by(species) |>
  summarise(avg_mass_g = mean(body_mass_g, na.rm = TRUE)) |>
  show_query()
```

Creating interactive web components

In week 8 we saw how reactive programming can add interactivity to web applications using shiny.

Today we'll see how to use R and Javascript to create interactive UI elements.

Concepts combined

The UI elements from today and the reactive server code from week 8 is all the ingredients to create shiny apps.

Shiny extensions

There are many JS libraries which have been wrapped up into R packages, for use in Shiny or regular analysis.

https://github.com/nanxstats/awesome-shiny-extensions

htmltools

```r
library(htmltools)
```

The htmltools package allows you to write HTML code with R.

```r
div(
  p("Hello world!"),
  img(src = "earth.jpg")
)
```

```html
<div>
  <p>Hello world!</p>
  <img src="earth.jpg"/>
</div>
```

Hello world!

htmltools

This is used to create the UI of a Shiny app.

It is also include the necessary CSS/JS dependencies for HTML reports with interactive ‘widgets’.

htmlwidgets

The htmlwidgets package provides a framework for creating R bindings to JavaScript libraries. HTML Widgets can be:

- Used at the R console for data analysis just like conventional R plots.
- Embedded within R Markdown documents
- Incorporated into Shiny web applications.
- Saved as standalone web pages for ad-hoc sharing via email, file transfer, web deployment, etc.

htmlwidgets showcase

The htmlwidgets package powers many popular R packages including:

- leaflet
- plotly
- visNetwork
- DiagrammeR

http://www.htmlwidgets.org/

htmlwidgets components

All widgets include the following components:

- Web dependencies: JS and CSS assets used by the widget
- R binding: This is the function that users call to create the output
- JavaScript binding: The JavaScript code that glues everything together, passing data/options from the R binding to the underlying JavaScript library.

htmlwidgets setup

From within a package, you can quickly get started with a htmlwidget using:

```r
htmlwidgets::scaffoldWidget("mywidget")
```

🔥 Follow along!

Create a package for making interactive wordclouds. We’ll use the wordcloud2.js library, available on GitHub here: https://github.com/timdream/wordcloud2.js

htmlwidgets setup

The htmlwidgets components are organised in packages with this file structure:

```
R/
| <name>.R

inst/
|-- htmlwidgets/
|   |-- <name>.js
|   |-- <name>.yaml
|   |-- lib/
|   |   |-- <javascript library>/
```

Web dependencies

Dependencies are specified using the YAML configuration file located at `inst/htmlwidgets/<name>.yaml`.

```yaml
dependencies:
  - name: <name>
    version: <version>
    src: htmlwidgets/lib/<src>
    script:
      - <JS files>
    stylesheet:
      - <CSS files>
```

Web dependencies

🔥 Follow along!

Download the JavaScript src for wordcloud2.js and add it to the package as a htmlwidgets dependency.
The JavaScript library's sources are available in the repository's `src/` folder.
https://github.com/timdream/wordcloud2.js

R binding

An R function which returns a htmltools widget created with `htmlwidgets::createWidget()`

```r
function(x, ...) {
  # R code preparing data/settings

  # Return a HTML widget
  createWidget(
    name, # The name of your widget in /inst
    x, # The data/settings for the widget's JS binding
    ...
  )
}
```

R binding

Follow along!

Update the generated R binding function to:

- Accept a character vector of words.
- Accept a numeric vector of frequency/weight.
- Pass these inputs into the htmlwidget via `x`.

Bonus: improve the design by accepting `.data` as the first input, then using tidy evaluation to pass in the words and frequencies from `.data`.

JavaScript binding

The JavaScript code that takes data/settings from R and uses the JS library to create the output.

```
HTMLWidgets.widget({
  name: "<name>",
  type: "output",
  factory: function(el, width, height) {
    // initialise the JavaScript object from the library here
    var obj = new <initalise object>;
    return {
      renderValue: function(x) {
        // update the initalised JavaScript object with new data/settings
      },
      resize: function(width, height) {
        // Re-render or otherwise update size when window changes
      }
    };
  }
```

JavaScript binding

Follow along!

Update the generated JavaScript binding to create the word-cloud on the htmlwidgets HTML element `el`.

Hint: a wordcloud is created using `wordcloud2.js` with:

```
WordCloud(el, { list: [['foo', 12], ['bar', 6]] } );
```

Hint: The data can be transposed from two separate vectors into the above format with `HTMLWidgets.transposeArray2D([x.words, x.freqs])`

Create a wordcloud

Your turn!

Your wordcloud function is now ready to use, try it out!

You can try it with the love words example dataset here:

```r
readr::read_csv(
  "https://arp.numbat.space/week11/lovewords.csv"
)
```

Use the wordclouds in a shiny app

The bindings for shiny apps are already created by `htmlwidgets::scaffoldWidget()`, and can be used in shiny like any other UI output and server renderer.

```r
widgetOutput <- function(outputId, width = '100%', height = '400px'){
  htmlwidgets::shinyWidgetOutput(outputId, 'widget', width, height,
                                 package = 'package')
}

renderWidget <- function(expr, env = parent.frame(), quoted = FALSE) {
  if (!quoted) { expr <- substitute(expr) } # force quoted
  htmlwidgets::shinyRenderWidget(expr, widgetOutput, env, quoted = TRUE)
}
```