Lab 08 - University of Edinburgh Art Collection

Tina Huynh

Contents

The University of Edinburgh Art Collection *"supports the world-leading research and teaching that happens within the University. Comprised of an astonishing range of objects and ideas spanning two millennia and a multitude of artistic forms, the collection reflects not only the long and rich trajectory of the University, but also major national and international shifts in art history."*[1].

```
See the sidebar [here](https://collections.ed.ac.uk/art) and note that there are 2970 pieces in the art
```

In this lab we'll scrape data on all art pieces in the Edinburgh College of Art collection.

1 Learning goals

- Working with R scripts
- Web scraping from a single page
- Writing functions
- Iteration by mapping functions
- Writing data out

[1]Source: https://collections.ed.ac.uk/art/about

2 Lab prep

- Download and install the SelectorGadget for your browser. Once you do, you should now be able to access SelectorGadget by clicking on the icon next to the search bar in your Chrome or Firefox browser.
  - See here for Chrome (recommended)
  - See here for Firefox
- Read the following on working with R Markdown documents vs. R scripts.

2.1 R scripts vs. R Markdown documents

Today we will be using both R scripts and R Markdown documents:

- `.R`: R scripts are plain text files containing **only** code and brief comments,
  - We'll use R scripts in the web scraping stage and ultimately save the scraped data as a csv.
- `.Rmd`: R Markdown documents are plain text files containing.
  - We'll use an R Markdown document in the web analysis stage, where we start off by reading in the csv file we wrote out in the scraping stage.

Here is the organization of your repo, and the corresponding section in the lab that each file will be used for:

3 Getting started

Go to the course GitHub organization and locate your homework repo, clone it in RStudio and open the R Markdown document. Knit the document to make sure it compiles without errors.

3.1 Warm up

Let's warm up with some simple exercises. Update the YAML of your R Markdown file with your information, knit, commit, and push your changes. Make sure to commit with a meaningful commit message. Then, go to your repo on GitHub and confirm that your changes are visible in your Rmd **and** md files. If anything is missing, commit and push again.

3.2 Packages

We'll use the **tidyverse** package for much of the data wrangling and visualisation, the **robotstxt** package to check if we're allowed to scrape the data, the **rvest** package for data scraping. These packages are already installed for you. You can load them by running the following in your Console:

```r
library(tidyverse)
library(robotstxt)
library(rvest)
```

3.3 Data

This assignment does not come with any prepared datasets. Instead you'll be scraping the data! But before doing so, let's check that a bot has permissions to access pages on this domain.

```
paths_allowed("https://collections.ed.ac.uk/art)")
```

```
##  collections.ed.ac.uk

## [1] TRUE
```

4 Exercises

4.1 Scraping a single page

Tip: To run the code you can highlight or put your cursor next to the lines of code you want to run

Work in `scripts/01-scrape-page-one.R`.

We will start off by scraping data on the first 10 pieces in the collection from here.

First, we define a new object called `first_url`, which is the link above. Then, we read the page at this url with the `read_html()` function from the **rvest** package. The code for this is already provided in `01-scrape-page-one.R`.

```
# set url
first_url <- "https://collections.ed.ac.uk/art/search/*:*/Collection:%22edinburgh+college+of+art%7C%7C%

# read html page
page <- read_html(first_url)
```

For the ten pieces on this page we will extract `title`, `artist`, and `link` information, and put these three variables in a data frame.

4.2 Titles

Let's start with titles. We make use of the SelectorGadget to identify the tags for the relevant nodes:

```
page %>%
  html_nodes(".iteminfo") %>%
  html_node("h3 a")
```

```
## {xml_nodeset (10)}
##  [1] <a href="./record/21405?highlight=*:*">Portrait of a Man                 ...
##  [2] <a href="./record/53492?highlight=*:*">Unknown                           ...
##  [3] <a href="./record/22690?highlight=*:*">Male Portrait                     ...
##  [4] <a href="./record/122805?highlight=*:*">The Sofa                         ...
##  [5] <a href="./record/99401?highlight=*:*">Untitled - Woman with Bun         ...
##  [6] <a href="./record/99974?highlight=*:*">Untitled                          ...
##  [7] <a href="./record/21673?highlight=*:*">Via D' Italia                     ...
##  [8] <a href="./record/50562?highlight=*:*">Unknown                           ...
##  [9] <a href="./record/21677?highlight=*:*">Northern Highland Landscape       ...
## [10] <a href="./record/99482?highlight=*:*">Untitled - City Highrise          ...
```

Then we extract the text with `html_text()`:

```
page %>%
  html_nodes(".iteminfo") %>%
  html_node("h3 a") %>%
  html_text()
```

```
##  [1] "Portrait of a Man                                                                          (
##  [2] "Unknown                                                                          (MAR 1960)"
##  [3] "Male Portrait                                                                                    (1969
##  [4] "The Sofa                                                                                    (1950)"
##  [5] "Untitled - Woman with Bun
##  [6] "Untitled                                                                                    (2007)"
##  [7] "Via D' Italia                                                                                    (Unkn
##  [8] "Unknown                                                                                    (1950)"
##  [9] "Northern Highland Landscape
## [10] "Untitled - City Highrise
```

And get rid of all the spurious white space in the text with `str_squish()`, which reduces repeated whitespace inside a string.

```
Take a look at the help for `str_squish()` to find out more about how it works and how it's different f
```

```
page %>%
  html_nodes(".iteminfo") %>%
  html_node("h3 a") %>%
  html_text() %>%
  str_squish()
```

```
##  [1] "Portrait of a Man (1954)"            "Unknown (MAR 1960)"
##  [3] "Male Portrait (1969)"                "The Sofa (1950)"
##  [5] "Untitled - Woman with Bun (1963)"    "Untitled (2007)"
##  [7] "Via D' Italia (Unknown)"             "Unknown (1950)"
##  [9] "Northern Highland Landscape (1934)"  "Untitled - City Highrise (Feb 1962)"
```

And finally save the resulting data as a vector of length 10:

```
titles <- page %>%
  html_nodes(".iteminfo") %>%
  html_node("h3 a") %>%
  html_text() %>%
  str_squish()
```

4.3 Links

The same nodes that contain the text for the titles also contains information on the links to individual art piece pages for each title. We can extract this information using a new function from the rvest package, `html_attr()`, which extracts attributes.

A mini HTML lesson! The following is how we define hyperlinked text in HTML:

```
<a href="https://www.google.com">Search on Google</a>
```

And this is how the text would look like on a webpage: Search on Google.

Here the text is `Search on Google` and the `href` attribute contains the url of the website you'd go to if you click on the hyperlinked text: `https://www.google.com`.

The moral of the story is: the link is stored in the `href` attribute.

```
page %>%
  html_nodes(".iteminfo") %>% # same nodes
  html_node("h3 a") %>% # as before
  html_attr("href") # but get href attribute instead of text
```

```
##  [1] "./record/21405?highlight=*:*"  "./record/53492?highlight=*:*"
##  [3] "./record/22690?highlight=*:*"  "./record/122805?highlight=*:*"
##  [5] "./record/99401?highlight=*:*"  "./record/99974?highlight=*:*"
##  [7] "./record/21673?highlight=*:*"  "./record/50562?highlight=*:*"
##  [9] "./record/21677?highlight=*:*"  "./record/99482?highlight=*:*"
```

These don't really look like URLs as we know then though. They're relative links.

```
See the help for `str_replace()` to find out how it works. Remember that the first argument is passed i
```

1. Click on one of art piece titles in your browser and take note of the url of the webpage it takes you to. Think about how that url compares to what we scraped above? How is it different? Using `str_replace()`, fix the URLs. You'll note something special happening in the `pattern` to replace. We want to replace the `.`, but we have it as `\\.`. This is because the period `.` is a special character and so we need to escape it first with backslashes, `\\`s.

```r
links <- page %>%
  html_nodes(".iteminfo") %>%
  html_node("h3 a") %>%
  html_attr("href") %>%
  str_replace("^", "https://collections.ed.ac.uk")
```

4.4 Artists

2. Fill in the blanks to scrape artist names. Use SelectorGadget to identify which CSS selector contains the artist names. Remember to change `eval = TRUE`.

```r
artists <- page %>%
  html_nodes(".iteminfo") %>%
  html_node("h4 a") %>%
  html_text() %>%
  str_squish()
```

4.5 Put it altogether

3. Fill in the blanks to organize everything in a tibble.

```r
first_ten <- tibble(
  title = titles,
  artist = artists,
  link = links
)
```

4.6 Scrape the next page

4. Click on the next page, and grab its url. Fill in the blank in to define a new object: `second_url`. Copy-paste code from top of the R script to scrape the new set of art pieces, and save the resulting data frame as `second_ten`.

If you haven't done so recently, c*ommit and push your changes to GitHub with an appropriate commit message. Make sure to commit and push all changed files so that your Git pane is cleared up afterwards.*

```r
# set url for second page
second_url <- "https://collections.ed.ac.uk/art/search/*:*/Collection:%22edinburgh+college+of+art%7C%7C

# read html page
page <- read_html(second_url)

# scrape titles
titles <- page %>%
  html_nodes(".iteminfo") %>%
  html_node("h3 a") %>%
  html_text() %>%
  str_squish()

# scrape links
links <- page %>%
  html_nodes(".iteminfo") %>%
  html_node("h3 a") %>%
  html_attr("href") %>%
  str_replace("^", "https://collections.ed.ac.uk")

# scrape artists
artists <- page %>%
  html_nodes(".iteminfo") %>%
  html_node("h4 a") %>%
  html_text() %>%
  str_squish()

# create data frame
second_ten <- tibble(
  title = titles,
  artist = artists,
  link = links
)
```

4.7 Functions

Work in `scripts/02-scrape-page-function.R`.

You've been using R functions, now it's time to write your own!

Let's start simple. Here is a function that takes in an argument `x`, and adds 2 to it.

```r
add_two <- function(x) {
  x + 2
}
```

Let's test it:

```r
add_two(3)
```

```
## [1] 5
```

```r
add_two(10)
```

```
## [1] 12
```

The skeleton for defining functions in R is as follows:

```r
function_name <- function(input) {
  # do something with the input(s)
  # return something
}
```

Then, a function for scraping a page should look something like:

```
**Reminder:** Function names should be short but evocative verbs.
```

```r
function_name <- function(url) {
  # read page at url
  # extract title, link, artist info for n pieces on page
  # return a n x 3 tibble
}
```

5. Fill in the blanks using code you already developed in the previous exercises. Name the function `scrape_page`.

Test out your new function by running the following in the console. Does the output look right? Discuss with teammates whether you're getting the same results as before.

```
scrape_page(first_url)
scrape_page(second_url)
```

If you haven't done so recently, c*ommit and push your changes to GitHub with an appropriate commit message. Make sure to commit and push all changed files so that your Git pane is cleared up afterwards.*

4.8 Iteration

Work in `scripts/03-scrape-page-many.R`.

We went from manually scraping individual pages to writing a function to do the same. Next, we will work on making our workflow a little more efficient by using R to iterate over all pages that contain information on the art collection.

```
**Reminder:** The collection has 2970 pieces in total.
```

That means we give develop a list of URLs (of pages that each have 10 art pieces), and write some code that applies the `scrape_page()` function to each page, and combines the resulting data frames from each page into a single data frame with 2970 rows and 3 columns.

4.9 List of URLs

Click through the first few of the pages in the art collection and observe their URLs to confirm the following pattern:

```
[sometext]offset=0     # Pieces 1-10
[sometext]offset=10    # Pieces 11-20
[sometext]offset=20    # Pieces 21-30
[sometext]offset=30    # Pieces 31-40
...
[sometext]offset=2960  # Pieces 2961-2970
```

We can construct these URLs in R by pasting together two pieces: (1) a common (`root`) text for the beginning of the URL, and (2) numbers starting at 0, increasing by 10, all the way up to 2970. Two new functions are helpful for accomplishing this: `glue()` for pasting two pieces of text and `seq()` for generating a sequence of numbers.

6. Fill in the blanks to construct the list of URLs.

4.10 Mapping

Finally, we're ready to iterate over the list of URLs we constructed. We will do this by **map**ping the function we developed over the list of URLs. There are a series of mapping functions in R (which we'll learn about in more detail tomorrow), and they each take the following form:

```
map([x], [function to apply to each element of x])
```

In our case `x` is the list of URLs we constructed and the function to apply to each element of `x` is the function we developed earlier, `scrape_page`. And as a result we want a data frame, so we use `map_dfr` function:

```r
map_dfr(urls, scrape_page)
```

7. Fill in the blanks to scrape all pages, and to create a new data frame called `uoe_art`.

4.11 Write out data

8. Finally write out the data frame you constructed into the `data` folder so that you can use it in the analysis section.

Aim to make it to this point during the workshop.

If you haven’t done so recently, c*ommit and push your changes to GitHub with an appropriate commit message. Make sure to commit and push all changed files so that your Git pane is cleared up afterwards.*

4.12 Analysis

Work in `lab-08.Rmd` for the rest of the lab.

Now that we have a tidy dataset that we can analyze, let’s do that!

We’ll start with some data cleaning, to clean up the dates that appear at the end of some title text in parentheses. Some of these are years, others are more specific dates, some art pieces have no date information whatsoever, and others have some non-date information in parentheses. This should be interesting to clean up!

First thing we’ll try is to separate the `title` column into two: one for the actual `title` and the other for the `date` if it exists. In human speak, we need to

> “separate the `title` column at the first occurrence of `(` and put the contents on one side of the `(` into a column called `title` and the contents on the other side into a column called `date`”

Luckily, there’s a function that does just this: `separate()`!

And once we have completed separating the single `title` column into `title` and `date`, we need to do further clean-up in the `date` column to get rid of extraneous `)`s with `str_remove()`, capture year information, and save the data as a numeric variable.

```
**Hint:** Remember escaping special characters from yesterday's lecture? You'll need to use that trick
```

9. Fill in the blanks in to implement the data wrangling we described above. Note that this will result in some warnings when you run the code, and that’s OK! Read the warnings, and explain what they mean, and why we are ok with leaving them in given that our objective is to just capture `year` where it’s convenient to do so.
10. Print out a summary of the data frame using the `skim()` function. How many pieces have artist info missing? How many have year info missing?
11. Make a histogram of years. Use a reasonable binwidth. Do you see anything out of the ordinary?

```
**Hint:** You'll want to use `mutate()` and `if_else()` or `case_when()` to implement the correction.
```

12. Find which piece has the out of the ordinary year and go to its page on the art collection website to find the correct year for it. Can you tell why our code didn’t capture the correct year information? Correct the error in the data frame and visualize the data again.

If you haven’t done so recently, knit, commit, and push your changes to GitHub with an appropriate commit message. Make sure to commit and push all changed files so that your Git pane is cleared up afterwards.

13. Who is the most commonly featured artist in the collection? Do you know them? Any guess as to why the university has so many pieces from them?

```
**Hint:** `str_subset()` can be helful here. You should consider how you might capture titles where the
```

14. Final question! How many art pieces have the word "child" in their title? Try to figure it out, and ask for help if you're stuck.

Knit, *commit, and push your changes to GitHub with an appropriate commit message. Make sure to commit and push all changed files so that your Git pane is cleared up afterwards and review the md document on GitHub to make sure you're happy with the final state of your work.*