Introduction to iteration

for loops, apply, and purrr

by Martin Frigaard

Written: January 09 2022

Updated: April 19 2023

https://github.com/paradigmdatagroup/iteration

Outline

When to use iteration

- Iteration & the DRY principle

Base R

- `for` loops
- Anonymous functions
- the `apply` family

`purrr`

- A `purrr` template
- `map()` variants
- Formula syntax

Worked examples

- Dealing with multiple datasets

What is iteration?

What is iteration?

In programming, iteration refers to *defining an input and applying an operation over every part.*

> "...across each of these, do this..."

How iteration works

The number of 'iterations' can be based on conditions, a set number, or via the number of elements in an object.

I like to this of iteration as,

"*versatile repetitive execution at scale*"

When to use iteration?

If the **DRY** principle is violated:

> *"Every piece of knowledge or logic should have a single, authoritative representation within a system."* - **D**on't **R**epeat **Y**ourself (Wikipedia)

When you find yourself copying and pasting code in multiple places, consider writing a function or using iteration

Problems to solve with iteration

- **Problems involving repetition**
    - Perform an operation needs on each element in a dataset
- **Problems involving conditional calculations**
    - Execute a set of calculations until a specific condition is met

- **Any combination of the two**
- Repeat an operation (or a set of operations) a certain number of times or until a specific condition is met

Methods for iteration in R

1. `for` loops (base R)

2. `apply` family of functions (base R)

3. `purrr` (`tidyverse`)

The `for` loop

The for loop structure

for loops are composed in three parts:

1. **A sequence to index**

2. **Operation(s) to iterate**

3. **An object to capture the results**

```r
# build output to capture result
output <- structure(list(words = NULL,
                         sentences = NULL,
                         letters = NULL))
# define sequence
for (variable x in my_list) {
  # list operation(s) to be repeated
  output[[x]] <- function(my_list[[x]])
}
```

The for loop example

Build a list

```r
my_list <- list(
  words = c("bLOW", "FOLloW", "cOMMOn",
            "ORiginAL", "UsUal"),
  sentences = c(
    "HE TaKeS tHE oatH OF offICe EaCH mArcH.",
    "THE OfficE pAINt waS A dUll, saD TAn.",
    "faRMers CAmE iN TO ThREsH tHe OAT crOP."
  ),
  letters = c("l", "y", "d", "p", "h",
              "e", "v", "M", "R", "Z")
)
```

View it's structure

```r
my_list
```

```
#> $words
#> [1] "bLOW"     "FOLloW"   "cOMMOn"
"ORiginAL" "UsUal"
#>
#> $sentences
#> [1] "HE TaKeS tHE oatH OF offICe EaCH mArcH."
#> [2] "THE OfficE pAINt waS A dUll, saD TAn."
#> [3] "faRMers CAmE iN TO ThREsH tHe OAT crOP."
#>
#> $letters
#>  [1] "l" "y" "d" "p" "h" "e" "v" "M" "R" "Z"
```

The `for` loop example

Apply a function to every element of a list

What happens when we pass `my_list` to `tolower()`?

```r
tolower(x = my_list)
```

```
#> [1] "c(\"blow\", \"follow\", \"common\", \"original\", \"usual\")"
#> [2] "c(\"he takes the oath of office each march.\", \"the office paint was a dull,
sad tan.\", \"farmers came in to thresh the oat crop.\")"
#> [3] "c(\"l\", \"y\", \"d\", \"p\", \"h\", \"e\", \"v\", \"m\", \"r\", \"z\")"
```

Yikes!

The `for` loop example

What happened?

```
??tolower
```

> *"`x` = a character vector, or an object that can be coerced to character"*

`tolower()` was expecting `x` to be a vector

The `for` loop example

A lot functions in R expect vectors, and a lot of vectors end up in lists...

What we wanted:

```
tolower(x = my_list$words)
#> [1] "blow"     "follow"   "common"   "original" "usual"
tolower(x = my_list$sentences)
#> [1] "he takes the oath of office each march."
#> [2] "the office paint was a dull, sad tan."
#> [3] "farmers came in to thresh the oat crop."
tolower(x = my_list$letters)
#>  [1] "l" "y" "d" "p" "h" "e" "v" "m" "r" "z"
```

The for loop sequence

Use seq_along() to define the sequence to index:

```r
# This generates a full sequence for my_list
seq_along(my_list)
```

```
#> [1] 1 2 3
```

```r
# This returns a single value of my_list
seq_along(my_list)[1]
```

```
#> [1] 1
```

```r
# This gets all items at index 1 in my_list
my_list[[seq_along(my_list)[1]]]
```

```
#> [1] "bLOW"     "FOLloW"   "cOMMOn"
"ORiginAL" "UsUal"
```

The for loop operations

The operations are the functions the for loop will perform per iteration

Test this with a few values

```
tolower(my_list[[1]])
```

```
#> [1] "blow"     "follow"   "common"   "original" "usual"
```

```
tolower(my_list[[3]])
```

```
#>  [1] "l" "y" "d" "p" "h" "e" "v" "m" "r" "z"
```

The `for` loop capture object

Define an object to capture the results of the loop

Make sure `output_list` is the same size as `my_list`

```r
vector(mode = "list", length = 3)
```

```
#> [[1]]
#> NULL
#> 
#> [[2]]
#> NULL
#> 
#> [[3]]
#> NULL
```

The `for` loop

Finally, we put it all together:

```r
# define capture object
output_list <- vector(mode = "list", length = 3)
# write sequence
for (x in seq_along(my_list)) {
  # write operations/capture in object
  output_list[[x]] <- tolower(my_list[[x]])
}
```

The for loop

The output:

```
output_list
```

```
#> [[1]]
#> [1] "blow"     "follow"   "common"   "original" "usual"
#> 
#> [[2]]
#> [1] "he takes the oath of office each march."
#> [2] "the office paint was a dull, sad tan."
#> [3] "farmers came in to thresh the oat crop."
#> 
#> [[3]]
#>  [1] "l" "y" "d" "p" "h" "e" "v" "m" "r" "z"
```

The for loop (clean up)

We can also clean up the output:

Named vectors in output_list:

```r
# define capture object
output_list <- vector(mode = "list", length = 3)
# write sequence
for (x in seq_along(my_list)) {
  # write operations/capture in object
  output_list[[x]] <- tolower(my_list[[x]])
  # clean up container
  names(output_list) <- c("words", "sentences",
"letters")
}
```

```r
output_list
```

```
#> $words
#> [1] "blow"     "follow"   "common"
"original" "usual"
#>
#> $sentences
#> [1] "he takes the oath of office each march."
#> [2] "the office paint was a dull, sad tan."
#> [3] "farmers came in to thresh the oat crop."
#>
#> $letters
#>  [1] "l" "y" "d" "p" "h" "e" "v" "m" "r" "z"
```

Recap `for` loops

1) Define the sequence to index

```
for ( x in seq_along( input_list ))
```

2) List the operations to iterate

```
function(input_list[[x]])
```

3) Build an object to capture the results

```
output_list <- vector(mode = "list", length = length(input_list))
```

Anonymous functions

Anonymous functions

- Anonymous functions are commonly used in iteration (`for` loops, `apply` functions, and `purrr`). R introduced a new shorthand anonymous function syntax in version 4.1.0:

"`\(x)  x + 1` is parsed as `function(x)  x + 1`"

Standard anonymous function:

```r
(function(x) tolower(x))("pIrAtES Ship")
```

```
#> [1] "pirates ship"
```

New shorthand anonymous syntax:

```r
(\(x) tolower(x))("pIrAtES Ship")
```

```
#> [1] "pirates ship"
```

The `apply` family

The apply functions

The base R *apply family of functions (apply(), lapply(), sapply(), vapply(), etc.) remove a lot of the 'book keeping' code in for loops

We'll focus on lapply() and sapply()

```
lapply(X, FUN, ...)

sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

lapply() for lists

lapply() (pronounced ‘l-apply’) works with lists and has only two required arguments:

1. X the object we want to iterate over

2. FUN being the function we want iterated

```
lapply(X = my_list, FUN = tolower)
```

```
#> $words
#> [1] "blow"     "follow"   "common"   "original"
"usual"
#>
#> $sentences
#> [1] "he takes the oath of office each march."
#> [2] "the office paint was a dull, sad tan."
#> [3] "farmers came in to thresh the oat crop."
#>
#> $letters
#>  [1] "l" "y" "d" "p" "h" "e" "v" "m" "r" "z"
```

sapply()

sapply() will attempt to simplify the result depending on the X argument:

If X is a list containing vectors where every element has the same length (and it's greater than 1), then sapply() returns a matrix:

```
str(my_list[1])
```

```
#> List of 1
#>  $ words: chr [1:5] "bLOW" "FOLloW" "cOMMOn"
"ORiginAL" ...
```

```
sapply(X = my_list[1], FUN = tolower)
```

```
#>      words
#> [1,] "blow"
#> [2,] "follow"
#> [3,] "common"
#> [4,] "original"
#> [5,] "usual"
```

sapply()

If a vector is passed to X where every element is length 1, then a vector is returned:

```
str(my_list[[1]])
```

```
#>  chr [1:5] "bLOW" "FOLLoW" "cOMMOn" "ORiginAL" "UsUal"
```

```
sapply(X = my_list[[1]], FUN = tolower)
```

```
#>       bLOW     FOLLoW     cOMMOn   ORiginAL      UsUal 
#>     "blow"   "follow"   "common" "original"    "usual" 
```

Note the names are preserved

`sapply()`

Finally, if `X` is a list where elements have a length greater than 1, then `sapply()` returns a list (making it identical to `lapply()`)

```r
identical(x = sapply(X = my_list, FUN = tolower),
          y = lapply(X = my_list, FUN = tolower))
```

```
#> [1] TRUE
```

This is because `sapply()` is a wrapper around `lapply()`, but has `simplify` and `USE.NAMES` set to `FALSE`

Anonymous functions with `*apply` functions

If we were to write the examples above using anonymous functions, they would look like this:

```r
identical(
  # standard
  x = lapply(X = my_list, FUN = tolower),
  # anonymous shorthand
  y = my_list |> lapply(\(x) tolower(x))
)
```

```
#> [1] TRUE
```

```r
identical(
  # standard
  x = sapply(X = my_list, FUN = tolower),
  # anonymous shorthand
  y = my_list |> sapply(\(x) tolower(x))
)
```

```
#> [1] TRUE
```

Recap `*apply()` functions

- The `*apply()` functions are more efficient than `for` loops because we can iterate over vectors *or* lists with **less code**

- One downside of `*apply` functions is they don’t play well with `data.frame`s or `tibble`s

- `*apply` functions also aren't very uniform. Each function has slight variations in their arguments and rules for return values

The purrr package

`purrr` template

A great way to start using it’s functions is with the method covered in Charlotte Wickham’s tutorial:

1. **Do it for one element**

2. **Turn it into a recipe**

3. **Use `purrr::map()` to do it for all elements**

purrr template: *do it for one element*

The goal with the first step is to get a minimal working example with a single element from the object you want to iterate over (with the function you want to iterate with)

```r
# subset an element from the list
? <- my_list[[?]]
# apply a function to extracted element
tolower(?)
```

```r
my_words <- my_list[['words']]
tolower(my_words)
```

```
#> [1] "blow"     "follow"   "common"
"original" "usual"
```

`purrr` template: *turn it into a recipe*

A standard `purrr` recipe defines `.x` (the object) and `.f` (the function), followed by any additional function arguments

```
.x = my_list, .f = tolower
```

`.x` = a list or atomic vector

`.f` = the function we want to apply over every element in `.x`

purrr template: *map() it across all elements*

The .x argument is the list or vector to iterate over, and .f is the function applied to every element of .x

```
purrr::map(.x = my_list, .f = tolower)
```

```
#> $words
#> [1] "blow"     "follow"   "common"   
"original" "usual"   
#> 
#> $sentences
#> [1] "he takes the oath of office each 
march."
#> [2] "the office paint was a dull, sad 
tan."
#> [3] "farmers came in to thresh the oat 
crop."
#> 
#> $letters
#>  [1] "l" "y" "d" "p" "h" "e" "v" "m" "r" 
"z"
```

Anonymous functions with `purrr`

When using `purrr::map()`, the object can be 'piped' to an anonymous function

This...

```
purrr::map(.x = my_list, .f = tolower)
```

...becomes this

```
my_list |> purrr::map(\(x) tolower(x))
```

```
# compare outputs
identical(
  # standard syntax
  x = purrr::map(.x = my_list,
                 .f = tolower),
  # shorthand anonymous function
  y = my_list |>
        purrr::map(\(x) tolower(x))
  )
```

```
#> [1] TRUE
```

`map()` variants

`map` vector functions

For vectors, `purrr` has a set of functions for each type

We'll be using `mixed_list`--a list with five different types of vectors--to explore the `map()` vector functions:

```
mixed_list <- list(booleans = c(FALSE, TRUE, FALSE, TRUE),
    integers = c(3L, 4L, 2L, 9L, 1L),
    doubles = c(3.041, 2.735, 2.987, 3.044, 2.95),
    strings = c("true", "depend", "client", "equal", "round"),
    dates = structure(c(19453, 19413, 19363), class = "Date"))
```

```
mixed_list |> names()
```

```
#> [1] "booleans" "integers" "doubles"  "strings"  "dates"
```

map vector functions

Test vectors in mixed_list by matching is.<type>() function

- map_lgl() returns a logical vector
- map_int() returns an integer vector
- map_dbl() returns a double vector

...note that dates are stored as double vectors

```
mixed_list |> purrr::map_lgl(\(x) is.logical(x))
```

```
#> booleans integers  doubles  strings    dates
#>     TRUE    FALSE    FALSE    FALSE    FALSE
```

```
mixed_list |> purrr::map_int(\(x) is.integer(x))
```

```
#> booleans integers  doubles  strings    dates
#>        0        1        0        0        0
```

```
mixed_list |> purrr::map_dbl(\(x) is.double(x))
```

```
#> booleans integers  doubles  strings    dates
#>        0        0        1        0        1
```

map vector functions

Test vectors in mixed_list by matching is.<type>() function

map_chr() returns a character vector with a warning

```
mixed_list |> purrr::map_chr(\(x) is.character(x))
```

```
#> booleans integers  doubles  strings    dates
#>  "FALSE"  "FALSE"  "FALSE"   "TRUE"  "FALSE"
```

```
#> Warning: Automatic coercion from logical to character was deprecated in purrr 1.0.0.
#> ℹ Please use an explicit call to `as.character()` within `map_chr()` instead.
#> Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```

map_vec()

The previous `purrr::map_raw()` function has been replaced with `purrr::map_vec()`, which "*simplifies to the common type of the output*"

```
mixed_list |> purrr::map_vec(\(x) is.character(x))
```

```
#> booleans integers   doubles   strings     dates
#>     FALSE     FALSE     FALSE      TRUE     FALSE
```

Note that the results are no longer characters (in `"quotes"`). The same is true when I test the dates:

```
mixed_list |> purrr::map_vec(\(x) lubridate::is.Date(x))
```

```
#> booleans integers   doubles   strings     dates
#>     FALSE     FALSE     FALSE     FALSE      TRUE
```

Worked Examples

https://github.com/paradigmdatagroup/iteration

Iteration examples

Use cases I've continuously encountered and used iteration to solve:

1) Downloading multiple files

- URLs might share a common domain, but varying paths:

2) Copying/renaming multiple files

- Batch rename and relocate files

3) Importing multiple files

- Import a local folder of data files into RStudio

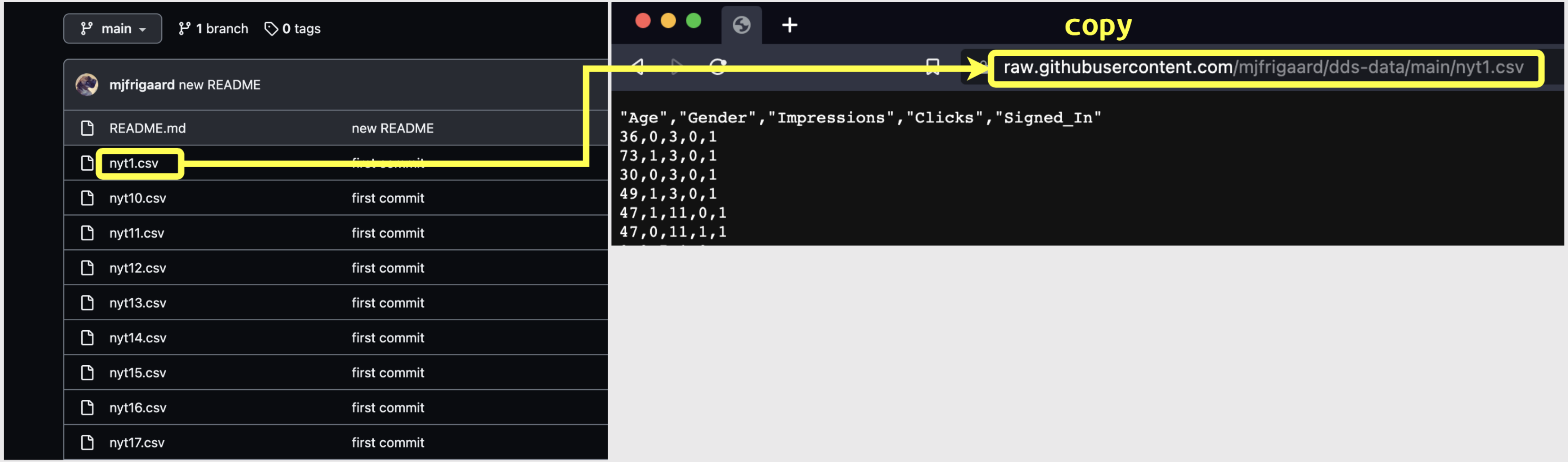
4) Exporting multiple objects

- Export multiple objects from RStudio into unique file paths

The files in these uses cases come from Doing Data Science by Cathy O'Neil and Rachel Schutt and are stored in this repository.

Downloading files

"I need to download multiple files (from separate URLS)" - Link to Github repo



mjfrigaard new README

README.md	new README
nyt1.csv	first commit
nyt10.csv	first commit
nyt11.csv	first commit
nyt12.csv	first commit
nyt13.csv	first commit
nyt14.csv	first commit
nyt15.csv	first commit
nyt16.csv	first commit
nyt17.csv	first commit

raw.githubusercontent.com/mjfrigaard/dds-data/main/nyt1.csv

```
"Age","Gender","Impressions","Clicks","Signed_In"
36,0,3,0,1
73,1,3,0,1
30,0,3,0,1
49,1,3,0,1
47,1,11,0,1
47,0,11,1,1
```

Downloading files

These files share a common domain, but have different file paths:

Step 1: Create unique URLS for one week (7) .csv files

- domain.com/path/to/file.csv

```r
# get example URL
nyt_url <- "https://raw.githubusercontent.com/mjfrigaard/dds-data/main/nyt1.csv"
# extract directory (i.e. common domain from URL)
nyt_dir_url <- fs::path_dir(nyt_url)
nyt_dir_url
```

```
#> [1] "https:/raw.githubusercontent.com/mjfrigaard/dds-data/main"
```

Downloading files

Step 1: Create unique URLS for the subset of .csv files

```r
# create file names for 7th through 13th
nyt_file_nms <- paste0("nyt", 7:13, ".csv")
head(nyt_file_nms, 3)
```

```
#> [1] "nyt7.csv" "nyt8.csv" "nyt9.csv"
```

```r
# combine domain with file name
nyt_file_urls <- paste(nyt_dir_url, nyt_file_nms, sep = "/")
head(nyt_file_urls, 3)
```

```
#> [1] "https:/raw.githubusercontent.com/mjfrigaard/dds-data/main/nyt7.csv"
#> [2] "https:/raw.githubusercontent.com/mjfrigaard/dds-data/main/nyt8.csv"
#> [3] "https:/raw.githubusercontent.com/mjfrigaard/dds-data/main/nyt9.csv"
```

Downloading files

Step 2: Create unique local folder and file paths for the .csv files:

```r
nyt_local_dir <- "dds-nyt"
# create folder
fs::dir_create(nyt_local_dir)
# create file paths
nyt_local_pths <- paste(nyt_local_dir, nyt_file_nms, sep = "/")
head(nyt_local_pths)
```

```
#> [1] "dds-nyt/nyt7.csv"  "dds-nyt/nyt8.csv"  "dds-nyt/nyt9.csv"
#> [4] "dds-nyt/nyt10.csv" "dds-nyt/nyt11.csv" "dds-nyt/nyt12.csv"
```

Downloading files

Step 3: Do it for one element of `nyt_file_urls` and `nyt_local_pths`:

```
download.file(url = nyt_file_urls[1], destfile = nyt_local_pths[1])
```

```
trying URL 'https:/raw.githubusercontent.com/mjfrigaard/dds-data/main/nyt7.csv'
Content type 'text/plain; charset=utf-8' length 4856135 bytes (4.6 MB)
==================================================
downloaded 4.6 MB
```

`download.file()` comes with a progress bar (more on that later)

Downloading files

We need a `purrr` function with the following arguments:

1. `x` = An input vector of existing url paths

2. `y` = The output vector of destination file paths

3. Any additional arguments for `download.file()`

For this problem, we don't need to assign a return value to an object…we need a `purrr` function that will iterate over the items in `x` and write them to the new location in `y`

Downloading files

`walk()` is ideal for problems like this:

> '*`walk()` returns the input `.x` (invisibly)*' ...and... '*the return value of `.f()` is ignored*'

`invisibly` = the output from a function doesn't need to be assigned to an object

`walk2()` because we have the file URLS (`nyt_file_urls`) *and* the local file paths (`nyt_local_pths`)

Downloading files

- We will also add `.progress = TRUE` to view `purrr`s progress bar (and `quiet = TRUE` to silence the `download.file()` progress bar).

```
purrr::walk2(
  .x = nyt_file_urls, .y = nyt_local_pths, # inputs
  .f =  download.file, # function
  .progress = TRUE, quiet = TRUE # additional arguments
  )
```

```
■■■■■■■■■■■■■■■■                   52% |  ETA: 15s
```

Progress bars!

Copying a directory of files

> *"I have a folder of files I'd like to rename or copy to a new directory"*

I just created this:

```
dds-nyt/
    ├── nyt10.csv
    ├── nyt11.csv
    ├── nyt12.csv
    ├── nyt13.csv
    ├── nyt7.csv
    ├── nyt8.csv
    └── nyt9.csv
```

but I'd like this:

```
dds-nyt
    └── raw
        ├── nyt10.csv
        ├── nyt11.csv
        ├── nyt12.csv
        ├── nyt13.csv
        ├── nyt7.csv
        ├── nyt8.csv
        └── nyt9.csv
```

Copying a directory of files

Create the new file paths

1) Store current file paths in vector

```r
# get file paths
file_pths <- list.files("dds-nyt", full.names = TRUE, pattern = ".csv$")
```

2) Do it for one (*replace the current path with desired folder*)

```r
# do it for one
gsub(pattern = "^dds-nyt", replacement = "dds-nyt/raw", x = file_pths[1])
```

```
#> [1] "dds-nyt/raw/nyt10.csv"
```

Copying a directory of files

Create the new file paths

3) Write recipe

```
# input
.x = file_pths,
# function
.f = gsub,
# args
pattern = "^dds-nyt",
replacement = "dds-nyt/raw"
```

`map_chr()` can apply `gsub()` across all `file_pths`

```
raw_file_pths <-  purrr::map_chr(
                      .x = file_pths,
                      .f = gsub,
                      pattern = "^dds-nyt",
                      replacement = "dds-nyt/raw")
head(raw_file_pths, 2)
```

```
#> [1] "dds-nyt/raw/nyt10.csv" "dds-nyt/raw/nyt11.csv"
```

Copying a directory of files

Now we're ready to copy the files

1) Do it for one

```
fs::dir_create("dds-nyt/raw")
# do it for one
fs::file_copy(
  path = file_pths[1],
  new_path = raw_file_pths[1],
  overwrite = TRUE)
fs::dir_tree("dds-nyt/raw", type = "any")
```

```
#> dds-nyt/raw
#> └── nyt10.csv
```

2) Write a recipe

```
# inputs
.x = file_pths, .y = raw_file_pths,
# function and args
.f = fs::file_copy, overwrite = TRUE
```

Copying a directory of files

walk2() it out!

```
purrr::walk2(.x = file_pths, .y = raw_file_pths,
      .f = fs::file_copy,
      .progress = TRUE, overwrite = TRUE
  )
fs::dir_tree("dds-nyt/raw", type = "any")
```

```
#> dds-nyt/raw
#> ├── nyt10.csv
#> ├── nyt11.csv
#> ├── nyt12.csv
#> ├── nyt13.csv
#> ├── nyt7.csv
#> ├── nyt8.csv
#> └── nyt9.csv
```

Copying a directory of files

What about the previous files in the parent `dds-nyt/` folder?

Supply the output from `list.files()` directly to `walk()` and include a pattern to matches `.csv` files

```
#> dds-nyt/
#> ├── nyt10.csv
#> ├── nyt11.csv
#> ├── nyt12.csv
#> ├── nyt13.csv
#> ├── nyt7.csv
#> ├── nyt8.csv
#> ├── nyt9.csv
#> └── raw
```

```
purrr::walk(
    # list CURRENT files
    .x = list.files(
      path = "dds-nyt",
      pattern = ".csv$",
      full.names = TRUE),
    # map function
    .f = fs::file_delete)
```

```
#> dds-nyt/
#> └── raw
#>     ├── nyt10.csv
#>     ├── nyt11.csv
#>     ├── nyt12.csv
#>     ├── nyt13.csv
#>     ├── nyt7.csv
#>     ├── nyt8.csv
#>     └── nyt9.csv
```

Import multiple datasets

> "*You'd like to import and combine several data files into a single dataset*"

Do it for one

```r
nyt1 <- vroom::vroom(
  file = raw_file_pths[1],
  delim = ",",
  show_col_types = FALSE)
```

```r
head(nyt1)
```

```
#> # A tibble: 6 × 5
#>     Age Gender Impressions Clicks Signed_In
#>   <dbl>  <dbl>       <dbl>  <dbl>     <dbl>
#> 1    59      1           4      0         1
#> 2     0      0           7      1         0
#> 3    19      0           5      0         1
#> 4    44      1           5      0         1
#> 5    30      1           4      0         1
#> 6    33      1           3      0         1
```

Import multiple datasets: add wrangle function

Add hypothetical wrangling steps to make this example more realistic

```r
nyt_data_processing <- function(nyt_csv) {
  orig_nms <- c("Age", "Gender", "Impressions", "Clicks", "Signed_In")
  nyt_nms <- names(nyt_csv)
  if (isFALSE(identical(x = orig_nms, y = nyt_nms))) {
    cli::cli_abort("these data don't have the correct columns!")
  } else {
    nyt_proc <- nyt_csv |> dplyr::mutate(age_group = dplyr::case_when(
            # create age_group variable
              Age < 18 ~ "<18", Age >= 18 & Age < 30 ~ "18-30", Age >= 30 & Age < 45 ~ "30-44",
              Age >= 45 & Age < 65 ~ "45-65", Age >= 65 ~ "65+"),
          # factor age_group (ordered)
          age_group = factor(age_group, levels = c("<18", "18-30", "30-44", "45-65", "65+"), ordered = TRUE),
          # create CTR variable
          ctr_rate = round(x = Clicks/Impressions, digits = 3),
          # create new Female variable
          female = dplyr::case_when(Gender == 0 ~ "yes", Gender == 1 ~ "no", TRUE ~ NA_character_),
          # factor female (un-ordered)
          female = factor(female, levels = c("no", "yes")),
          Signed_In = dplyr::case_when(Signed_In == 0 ~ "no", Signed_In == 1 ~ "yes", TRUE ~ NA_character_),
          # factor Signed_In (un-ordered) & format columns
          Signed_In = factor(Signed_In, levels = c("no", "yes"))) |> janitor::clean_names()
  }
  return(nyt_proc)
}
```

Import multiple datasets: set names

Get vector of raw data paths and set names (`purrr::set_names()`)

```
raw_data_pths <- purrr::set_names(x = list.files(path = "dds-nyt/raw",
                                                 pattern = ".csv$",
                                                 full.names = TRUE))
raw_data_pths |> head(2)
```

```
#>   dds-nyt/raw/nyt10.csv   dds-nyt/raw/nyt11.csv
#> "dds-nyt/raw/nyt10.csv" "dds-nyt/raw/nyt11.csv"
```

Setting names on the `raw_data_pths` vector will carry through to the imported list.

Import multiple datasets: import data

Add import function with `purrr::map()`

```r
# import
purrr::map(
  .x = raw_data_pths, .f = vroom::vroom, delim = ",", show_col_types = FALSE) |>
  # preview
  head(1) |> dplyr::glimpse()
```

Use `dplyr::glimpse()` to view the imported dataset in the list

Import multiple datasets: import data (preview)

`dplyr::glimpse()` shows us the original column names have been imported:

```
#> List of 1
#>  $ dds-nyt/raw/nyt10.csv: spc_tbl_ [452,766 × 5] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
#>   ..$ Age        : num [1:452766] 59 0 19 44 30 33 41 41 0 23 ...
#>   ..$ Gender     : num [1:452766] 1 0 0 1 1 1 0 0 0 1 ...
#>   ..$ Impressions: num [1:452766] 4 7 5 5 4 3 1 3 9 1 ...
#>   ..$ Clicks     : num [1:452766] 0 1 0 0 0 0 0 0 1 0 ...
#>   ..$ Signed_In  : num [1:452766] 1 0 1 1 1 1 1 1 0 1 ...
#>   ..- attr(*, "spec")=
#>   .. .. cols(
#>   .. ..   Age = col_double(),
#>   .. ..   Gender = col_double(),
#>   .. ..   Impressions = col_double(),
#>   .. ..   Clicks = col_double(),
#>   .. ..   Signed_In = col_double(),
#>   .. ..   .delim = ","
#>   .. .. )
#>   ..- attr(*, "problems")=<externalptr>
```

Import multiple datasets: wrangle

Add wrangling function with `purrr::map()`

```
# import
purrr::map(
  .x = raw_data_pths, .f = vroom::vroom, delim = ",", show_col_types = FALSE) |>
  # wrangle
  purrr::map(.f = nyt_data_processing) |>
  # preview
  head(1) |> dplyr::glimpse()
```

Import multiple datasets: wrangle (preview)

We can see the variables have been wrangled by the `nyt_data_processing()` function.

```
#> List of 1
#>  $ dds-nyt/raw/nyt10.csv: spc_tbl_ [452,766 × 8] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
#>   ..$ age        : num [1:452766] 59 0 19 44 30 33 41 41 0 23 ...
#>   ..$ gender     : num [1:452766] 1 0 0 1 1 1 0 0 0 1 ...
#>   ..$ impressions: num [1:452766] 4 7 5 5 4 3 1 3 9 1 ...
#>   ..$ clicks     : num [1:452766] 0 1 0 0 0 0 0 0 1 0 ...
#>   ..$ signed_in  : Factor w/ 2 levels "no","yes": 2 1 2 2 2 2 2 2 1 2 ...
#>   ..$ age_group  : Ord.factor w/ 5 levels "<18"<"18-30"<..: 4 1 2 3 3 3 3 3 1 2 ...
#>   ..$ ctr_rate   : num [1:452766] 0 0.143 0 0 0 0 0 0 0.111 0 ...
#>   ..$ female     : Factor w/ 2 levels "no","yes": 1 2 2 1 1 1 2 2 2 1 ...
#>   ..- attr(*, "spec")=
#>   .. .. cols(
#>   .. ..   Age = col_double(),
#>   .. ..   Gender = col_double(),
#>   .. ..   Impressions = col_double(),
#>   .. ..   Clicks = col_double(),
#>   .. ..   Signed_In = col_double(),
#>   .. ..   .delim = ","
#>   .. .. )
#>   ..- attr(*, "problems")=<externalptr>
```

Import multiple datasets: bind

For the final step, I'll bind all the data into a data.frame with the updated `purrr::list_rbind()` function (set `names_to = "id"`).

```r
# import
purrr::map(
  .x = raw_data_pths, .f = vroom::vroom, delim = ",", show_col_types = FALSE) |>
  # wrangle
  purrr::map(.f = nyt_data_processing) |>
  # bind
  purrr::list_rbind(names_to = "id") |>
  # preview
  dplyr::glimpse()
```

Import multiple datasets: bind (preview)

We can see the datasets from `dds-nyt/raw/` have been imported and processed.

```
#> Rows: 3,488,345
#> Columns: 9
#> $ id          <chr> "dds-nyt/raw/nyt10.csv", "dds-nyt/raw/nyt10.csv", "d…
#> $ age         <dbl> 59, 0, 19, 44, 30, 33, 41, 41, 0, 23, 28, 34, 0, 17,…
#> $ gender      <dbl> 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0…
#> $ impressions <dbl> 4, 7, 5, 5, 4, 3, 1, 3, 9, 1, 4, 4, 7, 3, 7, 6, 6, 2…
#> $ clicks      <dbl> 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0…
#> $ signed_in   <fct> yes, no, yes, yes, yes, yes, yes, yes, no, yes, yes,…
#> $ age_group   <ord> 45-65, <18, 18-30, 30-44, 30-44, 30-44, 30-44, 30-44…
#> $ ctr_rate    <dbl> 0.000, 0.143, 0.000, 0.000, 0.000, 0.000, 0.000, 0.0…
#> $ female      <fct> no, yes, yes, no, no, no, yes, yes, yes, no, no, no,…
```

Import multiple datasets: assign

Assign the imported/wrangled data to nyt_data_proc

```
# store
nyt_data_proc <- purrr::map(
    # import
    .x = raw_data_pths,
    .f = vroom::vroom,
    delim = ",",
    show_col_types = FALSE) |>
    # wrangle
  purrr::map(
    .f = nyt_data_processing) |>
    # bind
  purrr::list_rbind(names_to = "id")
```

```
nyt_data_proc |> dplyr::count(id)
```

```
#> # A tibble: 7 × 2
#>   id                            n
#>   <chr>                     <int>
#> 1 dds-nyt/raw/nyt10.csv 452766
#> 2 dds-nyt/raw/nyt11.csv 478066
#> 3 dds-nyt/raw/nyt12.csv 396308
#> 4 dds-nyt/raw/nyt13.csv 786044
#> 5 dds-nyt/raw/nyt7.csv  452493
#> 6 dds-nyt/raw/nyt8.csv  463196
#> 7 dds-nyt/raw/nyt9.csv  459472
```

id contains the name of the original file.

Export multiple datasets

> *"You'd like to split your data on a categorical variable into individual datasets, then export these into separate file paths"*

Now that we've imported and wrangled the data, we want to export these to a different location (i.e., `dds-nyt/processed/`) and not back in `dds-nyt/raw/`.

Creating a vector of processed data file paths is a little more involved because I wanted to add a date prefix to the exported files, and because I want to add this path as *a variable in the* `nyt_data_proc` *dataset*.

Export multiple datasets: add processed file names

Below I create file_nm and proc_file_pth

```
# create file names
nyt_data_proc <- dplyr::mutate(.data = nyt_data_proc,
        file_nm = tools::file_path_sans_ext(base::basename(id)),
        proc_file_pth = paste0("dds-nyt/processed/", as.character(Sys.Date()), "-", file_nm))
nyt_data_proc |> dplyr::count(proc_file_pth)
```

```
#> # A tibble: 7 × 2
#>   proc_file_pth                             n
#>   <chr>                                 <int>
#> 1 dds-nyt/processed/2023-04-19-nyt10 452766
#> 2 dds-nyt/processed/2023-04-19-nyt11 478066
#> 3 dds-nyt/processed/2023-04-19-nyt12 396308
#> 4 dds-nyt/processed/2023-04-19-nyt13 786044
#> 5 dds-nyt/processed/2023-04-19-nyt7  452493
#> 6 dds-nyt/processed/2023-04-19-nyt8  463196
#> 7 dds-nyt/processed/2023-04-19-nyt9  459472
```

Export multiple datasets: method 1

Note that I don’t include the file extension in `proc_file_pth` (*because I might want to use different file types when I’m exporting*).

In this first method, I’ll use the `base::split()` function to split `nyt_data_proc` by the `proc_file_pth` variable into a list of data frames. I’ll also use `utils::head()`, `purrr::walk()`, and `dplyr::glimpse()` to view the output.

```
split(x = nyt_data_proc, f = nyt_data_proc$proc_file_pth) |>
  utils::head(2) |>
  purrr::walk(.f = dplyr::glimpse)
```

Export multiple datasets: method 1 (preview)

```
#> Rows: 452,766
#> Columns: 11
#> $ id            <chr> "dds-nyt/raw/nyt10.csv", "dds-nyt/raw/nyt10.csv", "dds-n…
#> $ age           <dbl> 59, 0, 19, 44, 30, 33, 41, 41, 0, 23, 28, 34, 0, 17, 33,…
#> $ gender        <dbl> 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0,…
#> $ impressions   <dbl> 4, 7, 5, 5, 4, 3, 1, 3, 9, 1, 4, 4, 7, 3, 7, 6, 6, 2, 7,…
#> $ clicks        <dbl> 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
#> $ signed_in     <fct> yes, no, yes, yes, yes, yes, yes, yes, no, yes, yes, yes…
#> $ age_group     <ord> 45-65, <18, 18-30, 30-44, 30-44, 30-44, 30-44, 30-44, <1…
#> $ ctr_rate      <dbl> 0.000, 0.143, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, …
#> $ female        <fct> no, yes, yes, no, no, no, yes, yes, yes, no, no, no, yes…
#> $ file_nm       <chr> "nyt10", "nyt10", "nyt10", "nyt10", "nyt10", "nyt10", "n…
#> $ proc_file_pth <chr> "dds-nyt/processed/2023-04-19-nyt10", "dds-nyt/processed…
#> Rows: 478,066
#> Columns: 11
#> $ id            <chr> "dds-nyt/raw/nyt11.csv", "dds-nyt/raw/nyt11.csv", "dds-n…
#> $ age           <dbl> 28, 51, 29, 20, 19, 0, 58, 42, 35, 44, 62, 20, 0, 0, 43,…
#> $ gender        <dbl> 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
#> $ impressions   <dbl> 8, 5, 2, 4, 5, 3, 5, 6, 8, 4, 6, 4, 5, 4, 4, 5, 3, 2, 5,…
#> $ clicks        <dbl> 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2,…
#> $ signed_in     <fct> yes, yes, yes, yes, yes, no, yes, yes, yes, yes, yes, ye…
#> $ age_group     <ord> 18-30, 45-65, 18-30, 18-30, 18-30, <18, 45-65, 30-44, 30…
#> $ ctr_rate      <dbl> 0.000, 0.000, 0.000, 0.000, 0.000, 0.333, 0.200, 0.000, …
#> $ female        <fct> no, yes, no, no, yes, yes, yes, yes, no, yes, yes, yes, …
#> $ file_nm       <chr> "nyt11", "nyt11", "nyt11", "nyt11", "nyt11", "nyt11", "n…
#> $ proc_file_pth <chr> "dds-nyt/processed/2023-04-19-nyt11", "dds-nyt/processed…
```

Export multiple datasets: method 1 (prep)

Pass list to `purrr::walk2()` and iterate `vroom::vroom_write()` over processed data paths (`proc_file_pth`)

1) create processed data folder

```
fs::dir_create("dds-nyt/processed/")
```

2) create the `.x`, the split list of `nyt_data_proc` by `proc_file_pth`

```
by_proc_pths <- nyt_data_proc |>
  split(nyt_data_proc$proc_file_pth)
```

3) get unique processed data paths in `proc_file_pth` column and store as vector `.y`

```
proc_pths <-
paste0(unique(nyt_data_proc$proc_file_pth),
".csv")
```

Export multiple datasets: method 1 (export)

I can export the data to `proc_pths` using the standard syntax:

```r
# iterate with .f
purrr::walk2(.x = by_proc_pths, .y = proc_pths, .f = vroom::vroom_write, delim = ",")
```

Or with pipes as an anonymous function:

```r
nyt_data_proc |>
  split(nyt_data_proc$proc_file_pth) |>
  purrr::walk2(.y = proc_pths,
    \(x, y)
    vroom::vroom_write(x = x,
      file = y,  delim = ","))
```

Export multiple datasets: verify

```r
fs::dir_tree("dds-nyt/", pattern = "csv$")
```

```
#> dds-nyt/
#> ├── processed
#> │   ├── 2023-04-19-nyt10.csv
#> │   ├── 2023-04-19-nyt11.csv
#> │   ├── 2023-04-19-nyt12.csv
#> │   ├── 2023-04-19-nyt13.csv
#> │   ├── 2023-04-19-nyt7.csv
#> │   ├── 2023-04-19-nyt8.csv
#> │   └── 2023-04-19-nyt9.csv
#> └── raw
#>     ├── nyt10.csv
#>     ├── nyt11.csv
#>     ├── nyt12.csv
#>     ├── nyt13.csv
#>     ├── nyt7.csv
#>     ├── nyt8.csv
#>     └── nyt9.csv
```

Export multiple datasets: option 2

Another option involves the `group_walk()` function from `dplyr` (**WARNING**: this is experimental).

```r
nyt_data_proc |>
  dplyr::group_by(proc_file_pth) |>
  dplyr::group_walk( ~vroom::vroom_write(x = .x,
                          file = paste0(.y$proc_file_pth, ".csv"),
                          delim = ","))
```

Re-written as an anonymous function, this would look like:

```r
nyt_data_proc |>
  dplyr::group_by(proc_file_pth) |>
  dplyr::group_walk(\(x, y)
    vroom::vroom_write(
    x = x, file = paste0(y$proc_file_pth, ".csv"), delim = ", "))
```

Recap

- Iteration
  - What is iteration & what kinds of problems it can solve
- Base R
  - The structure of `for` loops & the `apply` family
    - New shorthand anonymous function syntax
- `purrr`
  - Creating a `purrr` template
  - `map()` variants (`map_vec()`)
- Worked examples
  - Dealing with multiple datasets
    - Downloading
    - Copying
    - Importing
    - Exporting

Read more

- `purrr` package website
- Iteration chapter in R for data science
- `purrr` version 1.0 blog post and video from Posit