Functional programming in R

With the purrr package

Derek Hansen

9/18/2019

What is Functional Programming?

"Functional programming is a programming paradigm . . . that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data." - Wikipedia

- Functions are like those in mathematics:
  1. They always return the same output for a given input
  2. No side-effects (e.g. modification of global variables)
  3. Functions that satisfy (1-2) are called **pure**
- Functions are **first-class** objects which can be passed as arguments to other functions (a.k.a. **functionals**)
- There is no changing state as the program runs; values are assigned once as functions of other values and inputs
- Everyday example: Microsoft Excel (without any VBA scripting!)

Why use functional programming?

- Through avoiding mutable state and composing pure functions, an FP program is also a pure function of its input
- This makes FP programs...
  - Modular
  - Predictable
  - Easier to test
  - Avoid common pitfalls involving changing state (e.g. global variables)

Why use functional programming?

Natural functionals in the FP paradigm include ...

- **map**: (where $f : X \to Y$)

$$((x_1, ..., x_n), f) \to (f(x_1), ..., f(x_n)))$$

- **filter**: (where f is a **predicate** function $f : X \to \{0, 1\}$)

$$((x_1, ..., x_n), f) \to (x_i : f(x_i) = 1)$$

- **reduce**: (where f is an **operator** function $f : X \times X \to X$)

$$((x_1, ..., x_n), f) \to f(x_1, f(x_2, f(x_3, f(...))))$$

Functional Programming in R

- R is **multi-paradigm**: it does not strictly adhere to FP principles, but it offers capability to use FP patterns
- Examples in base R include:
  - `Map`, `lapply`, `sapply`, `apply`, `vapply`, `mapply`
  - `Reduce`
  - `Filter`
- The `purrr` package by Hadley Wickham et al improves the the functional programming tools to R which are **syntactically consistent** and **type-safe**.

Mapping

- map is pretty much equivalent to lapply, but has some additional features

```
library(purrr)
my_sqrt <- function(x) sqrt(x)
str(map(c(1,2,3,4,5), my_sqrt))
```

```
## List of 5
##  $ : num 1
##  $ : num 1.41
##  $ : num 1.73
##  $ : num 2
##  $ : num 2.24
```

```
str(lapply(c(1,2,3,4,5), my_sqrt))
```

```
## List of 5
##  $ : num 1
##  $ : num 1.41
##  $ : num 1.73
```

Mapping

- If we want an atomic double vector instead of a list, the `map_dbl` ensures we always receive that.
- `sapply` does the same thing in this particular instance, but we can run into problems...

```
str(map_dbl(c(1,2,3,4,5), my_sqrt))
```

```
##  num [1:5] 1 1.41 1.73 2 2.24
```

```
str(sapply(c(1,2,3,4,5), my_sqrt))
```

```
##  num [1:5] 1 1.41 1.73 2 2.24
```

Problem: sapply is not type-safe!

- Example: Our colleague worked hard to make `my_sqrt` handle any real number.
- They even overwrote the function `my_sqrt` to make the transition seamless!

```
sqrt_general <- function(x) {
  if(x >= 0) sqrt(x)
  else return(paste0(sqrt(abs(x)), "i"))
}

my_sqrt <- sqrt_general
my_sqrt(5)
```

```
## [1] 2.236068
```

```
my_sqrt(-5)
```

```
## [1] "2.23606797749979i"
```

Problem: sapply is not type-safe!

```
str(sapply(c(1,2,3,4,5), my_sqrt))
```

```
##  num [1:5] 1 1.41 1.73 2 2.24
```

```
str(sapply(c(-1,2,-3,4,5), my_sqrt))
```

```
##  chr [1:5] "1i" "1.4142135623731" "1.73205080756888i" "2
```

- This is a great way to propogate errors. We have no way to guarentee whether sapply will return a "double" vector or a "string" vector.

map_dbl is type-safe!

```
str(map_dbl(c(1,2,3,4,5), my_sqrt))
```

```
##  num [1:5] 1 1.41 1.73 2 2.24
```

```
try(str(map_dbl(c(-1,2,-3,4,5), my_sqrt)))
```

```
## Error : Can't coerce element 1 from a character to a dou
```

- The `map_*` family of functions allows us to explictly impose which type we expect the output vector to be.
  - They "return an atomic vector of the indicated type (or die trying)" (documentation)

map_chr is type-safe!

```
map_chr(c(1,2,3,4,5), my_sqrt)
```

```
## [1] "1.000000" "1.414214" "1.732051" "2.000000" "2.23606
```

```
map_chr(c(-1,2,-3,4,5), my_sqrt)
```

```
## [1] "1i"                  "1.414214"            "1.73205080
## [4] "2.000000"            "2.236068"
```

- Suppose our colleague convinced our team lead that we should work exclusively with strings to accomodate complex numbers
- We use `map_chr` to reflect that now we want the output to be a character vector.
- No errors now because both doubles and characters can be coerced to double.

map_* is type-safe!

- `sapply` implicitly coerces to an atomic vector in the most general unit in the output for "convenience", but this is very prone to unexpected errors.
- Most of the time, it is better to be explicit to catch any errors early and keep type stability.
- Can also use `_lgl` for logical, `_int` for integer, `_raw` for raw type, `_dfr` and `_dfc` for data-table columns and rows.

Some more cool features of map - anonymous functions

- Can construct function in the argument using symbol notation

```
map_dbl(c(1,2,3,4,5), ~.x^2 + .x + sin(.x))
```

```
## [1]  2.841471  6.909297 12.141120 19.243198 29.041076
```

Some more cool features of map - multiple arguments

- Can use `map2_*` for 2 argument functions; `pmap_*` for n-argument functions
- The *i*th positional argument can be referenced with `..i` syntax.

```
map2_dbl(c(1,2,3,4,5), c(5,6,8,9,11), ~.x^2 + .y^2 + sin(.x
```

```
## [1]  26.84147  40.90930  73.14112  96.24320 145.04108
```

```
pmap_dbl(list(1:5, 11:15, 21:25), ~..1 + ..2 + ..3)
```

```
## [1] 33 36 39 42 45
```

```
pmap_dbl(list(1:5, 11:15, 21:25), function(x,y,z) x+y+z)
```

```
## [1] 33 36 39 42 45
```

Some more cool features of map - imap

- Can use `imap` if the names of the input list/vector are important.
- `imap_*(x, f(x,y))` is equivalent to `map2_*(x, names(x), f(x,y))`
- The type `dfr` indicates that we expect the function to output a **D**ata**F**rame **R**ow, which are then bound row-wise into a single dataframe.

```
library(dplyr)
midterm_grades <- c(Dan = 100, Derek = 20, Rob = 100)
grade_tbl      <- imap_dfr(midterm_grades, ~tibble(name =
grade_tbl
```

```
## # A tibble: 3 x 3
##   name  grade pass
##   <chr> <dbl> <lgl>
## 1 Dan     100 TRUE
## 2 Derek    20 FALSE
## 3 Rob     100 TRUE
```

Some more cool features of map - map_if

- `map_if` allows for use of a predicate function (or a vector) to only apply to certain values.
- It always returns a list (since the input and output could be of different types).

```
str(map_if(midterm_grades, !grade_tbl$pass, ~NA_real_))
```

```
## List of 3
##  $ Dan  : num 100
##  $ Derek: num NA
##  $ Rob  : num 100
```

```
str(map_if(midterm_grades, ~.x <= 50, ~"FAIL!!"))
```

```
## List of 3
##  $ Dan  : num 100
##  $ Derek: chr "FAIL!!"
##  $ Rob  : num 100
```

Some more cool features of map - map_if

- `modify_if` is the same as `map_if`, but enforces that the type is the same as the input

```
str(modify_if(midterm_grades, ~.x <= 50, ~NA_real_))
```

```
##  Named num [1:3] 100 NA 100
##  - attr(*, "names")= chr [1:3] "Dan" "Derek" "Rob"
```

```
try(str(modify_if(midterm_grades, ~.x <= 50, ~"FAIL!!")))
```

```
## Error : Can't coerce element 1 from a character to a dou
```

keep and discard

```
## Only keep students who passed
keep(midterm_grades, ~.x >= 50)
```

```
## Dan Rob
## 100 100
```

```
## Remove students who passed to get a list of students on
discard(midterm_grades, grade_tbl$pass)
```

```
## Derek
##    20
```

purrr in the wild - succinctly extract results from different models

```
library(dplyr)
library(magrittr)
aic_bic_tbl <- list(
  `Binary Poverty Indicator Interaction` = logis_res_census
  `Poverty Rate Interaction` = logis_res_census,
  `Income Interaction` = logis_res_census_inc_interact,
  `No Income` = logis_res_census_noincome,
  `No Poverty Rate` = logis_res_census_nopoor
) %>%
  map2_dfr(names(.), ~tibble(model = .y, aic = AIC(.x), bic
  arrange(aic)
aic_bic_tbl
```

- Example directly from my applied qual. (Could have used `imap_dfr`!)
- `purrr` was designed by the same authors as `dplyr` and plays nicely with other `tidyverse` functions (including the pipe object `%>%`)

purrr in the wild - reduce to best model

```
best_model <- list(
  `Binary Poverty Indicator Interaction` = logis_res_census
  `Poverty Rate Interaction` = logis_res_census,
  `Income Interaction` = logis_res_census_inc_interact,
  `No Income` = logis_res_census_noincome,
  `No Poverty Rate` = logis_res_census_nopoor
) %>%
  reduce(~ifelse(BIC(.x) < BIC(.y), .y, .x))
```

- `reduce` function applies an operator function to reduce a vector to one value
- Illustrating example, but in reality it would be more efficient to use `which.max(aic_bic_tbl$bic)` (because it uses C code and more efficient algorithm)

Conclusions

- Through the Functional Programming (FP) paradigm, `purrr` allows for more concise and error-robust R coding patterns
- Allows complex operations to be composed from simple building blocks by operating on user-specified functions
- Many, many more features are contained in `purrr` beyond what was shown today

Further reading

- Tidyverse website (https://purrr.tidyverse.org/)
- "Iteration" chapter in R for Data Science (https://r4ds.had.co.nz/iteration.html)
- Hadley's `plyr` package which handles array and data.frame inputs.

Thank You!