Functional Programming: What, why and how?

Tom Ratford - Veramed

PHUSE EU Connect 2022

Introduction

As we move towards a multilingual world, tried and tested SAS problem solving skills will be exposed to new data structures. As programmers we must learn new ways to think about problems. One of the paradigms available is functional programming, a style grounded in mathematics and reproducibility. It is a style of programming that is possible in R, Python and Julia (to name a few) and directs programmers to write efficient, clean and reproducible code by design. In this paper I will explore what makes a programming language functional, why we should use these new features and how you can practically apply them to common ADaM dataset and TFL programming problems in R.

Imperative programming

To understand functional programming we must first define imperative programming. This is the style of programming which we are used to as SAS programmers. Imperative programming focuses on the concept of 'state'. A change of state caused by code is known as a *side effect*. Consider a simple SAS program.

```
data mydata;
  set olddata;
  a=1;
run;
```

In the above example we have changed the state of the work library to include our new `mydata` dataset. The *side effect* of `data` is the creation of `mydata`. `set` does not have a side effect, it simply returns `olddata`. In the `mydata` dataset we changed the state of `a`, as a side effect of the line `a=1;`.

State is a very important aspect of programming imperatively. We enforce rules and guidelines to ensure that our side effects follow a logical pattern of execution. For example: ensuring programs can run 'in batch' from start to finish without errors. Consider the below.

```
proc sort data=olddata;
  by ordvar;
run;
```

This is generally considered bad programming practice as we change the state of `olddata` without adding an additional dataset for traceability.

Functional programming

Conversely, functional programming rejects the idea of state and instead focuses on input and output. In pure functional programming languages, we cannot access anything without it being passed to a function. System information such as the time or date cannot be accessed as these are constantly changing states. Hence it is commonly said that a pure functional programming language is useless, so we generally accept a looser definition.

> A language is considered functional if it directs or forces you to create a solution that does not rely on state.

Lets say we want to apply some function g, followed by another function f, on our data denoted x. In functional programming this is represented as composing f and g together.

$$f \circ g(x) = f(g(x))$$

You may recognise this as mathematics. As mentioned earlier a purely functional language is not completely practical. R, Python and Julia are not purely functional languages, but are instead considered multi-paradigm languages. This is because they do not force you down a particular style of programming, yet they do contain crucial elements to allow us to program functionally.

First class functions

A first class function essentially means that a function can be considered as a value like `1` or `"Hello"`.

Consider the `nchar` function in R, that returns the length of a string.

```
nchar
```

```
## function (x, type = "chars", allowNA = FALSE, keepNA = NA)
## .Internal(nchar(x, type, allowNA, keepNA))
## <bytecode: 0x0000000024c71968>
## <environment: namespace:base>
```

```
strlength <- nchar
strlength
```

```
## function (x, type = "chars", allowNA = FALSE, keepNA = NA)
## .Internal(nchar(x, type, allowNA, keepNA))
## <bytecode: 0x0000000024c71968>
## <environment: namespace:base>
```

Notice that `strlength` and `nchar` have the same definition.

```
nchar("Hello")
```

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

```
strlength("Hello")
```

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

First class functions in R also make sense when you consider the definition of a function in R. Which involves the assignment operator `<-`.

```
myFunc <- function(...) {
  ...
}
```

First class functions are a must in functional programming. This is because they lead into another concept known as *higher-order functions*.

Higher order function

A higher order function is a function which does at least one of the following:

- Takes a function as an argument
- Returns a function as its result

This is a confusing concept, but you've probably already used and encountered this without realising. When using packages such as `dplyr`, you have likely used the `filter` function. This is a functional programming staple, which we will implement ourselves in the next section

A function that takes a function as an argument Lets consider a case where we want to be able to subset a large vector of numbers (1 to 2000), based off whether the number is a *narcissistic number*. A narcissistic (or Armstrong) number is a number b in which the number is equal to the sum of its own digits raised to the power of the number of digits. For example.

$$\begin{aligned} 153 &= 1^3 + 5^3 + 3^3 \\ 370 &= 3^3 + 7^3 + 0^3 \\ 371 &= 3^3 + 7^3 + 1^3 \\ &\dots \\ 1643 &= 1^4 + 6^4 + 4^4 + 3^4 \end{aligned}$$

This problem could be done using some subsetting like `x[as.character(nchar(...))]` but it would very quickly get messy. We could also make multiple variables to do this imperatively. But we are doing functional programming, so lets make a function that first works out if a number is an narcissistic number.

```
is.narc <- function(x) { #x a number
  x_char <- as.character(x)
  nums <- strsplit(x_char, split="")[[1]]
  x == sum(as.numeric(nums)^length(nums))
}
```

We can quickly check it catches what we expect

```
is.narc(153)
```

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

```
is.narc(154)
```

```
## [1] FALSE
```

Unfortunately, this function only works on a single value, so we need a function which can check over a whole vector and keep only what we want.

```
myFilter <- function(v, f) { # v a vector, f a function which returns a boolean
  w <- rep(NA, length(v))
  for (i in seq_along(v)) {
    if (f(v[i])) {
      w[i] <- v[i]
    }
  }
  w[!is.na(w)]
}
```

The function `f` which returns a boolean is known as a *predicate*.

We can now find all narcissistic numbers up to 2000.

```
myFilter(1:2000,is.narc)
```

```
##  [1]    1    2    3    4    5    6    7    8    9  153  370  371  407 1634
```

This is very similar to when you use a `filter` or `subset` command. They just provide an easier syntax to perform these operations.

A function that returns a function as its result We can also write a function which can return a function which we can then call. Consider $f \circ g(x)$ mentioned earlier, we can create a function which returns this composition as a single function.

```
compose <- function(f,g) { # f & g both functions
  function(x) f(g(x))
}
r <- compose(sin, cos)
print(r(.5))
```

```
## [1] 0.7691964
```

Note in the `compose` function we did not specify a name for our returned function.

Anonymous functions

An anonymous function is a function definition that is not bound to any identifier (i.e. we don't save it to an object). Anonymous functions are also commonly referred to as *lambdas* (especially in Python) and sometimes *closures*. Technically closures are different to anonymous functions, as closures also capture the state of their surrounding environment. As - within R - everything is bounded by an environment, it is fair to say that all functions are closures (there are exceptions). However, in general the term 'closure' will be used to indicate that this is a function within a function. Consider our `myFilter` function from earlier. We can provide a function without defining it previously.

```
myFilter(1:20, function(x) x %% 3 == 0)
```

```
## [1]  3  6  9 12 15 18
```

Or equivalently in R 4.1 or newer.

```
myFilter(1:20, \(x) x %% 3 == 0)
```

Benefits over imperative programming

Functional programming prioritises immutability. This also implies reproducability, as in theory providing the same arguments (which are not modified by the function) to a function will always return the same value.

Applications in R

Base R vs the Tidyverse [`purrr`]

A crucial elements of functional programming is the concept of a *map* function. This is a function which performs a function over a list of inputs. In base R this is the `lapply`, `sapply` or `vapply` function.

```
lapply(1:3, function(x) x + 1)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] 4
```

`vapply` is an interesting function as we have to provide a parameter `FUN.VALUE` of the type we wish to return. It will then throw an error if this type is not returned, meaning we can write more defensive code using it. For mapping over a data frame base R has `apply`, and for mapping over multiple inputs we have `mapply`.

The tidyverse equivalent is the `purrr` R package. This package's aim is to improve R's functional programming tools by providing a more consistent and expansive set of tools. Base R `lapply/vapply` is rewritten as `purrr::map`. Anonymous functions can also be written using a simplified syntax. It uses `~` to represent the start of a function.

```
map(1:3, ~ . + 1)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] 4
```

Instead of specifying parameters yourself, they have been defined for you:

- `.` to represent a single input.
- `.x` and `.y` to represent 2 inputs.
- `..1`,`..2`,... to represent more inputs.

`map` also has the same ability as `vapply` to return a vector of a desired type using alternative functions `map_chr`, `map_dbl` or similar.

Performance wise, base R's `lapply` & equivalent are faster than `map` but `purrr` is more consistent. One such feature of `purrr` is that `map(list, <n>)` returns the n^{th} element from the input list. in base R this would be done with `lapply(list, function(x) x[[2]])`. This is useful when combined with pipes, where using the `[ ]` or `$` operators would be more problematic.

Consider getting the max of means of some randomly sampled data.

```
x <- list(rnorm(20),rnorm(20),rnorm(20)) %>%
  map(summary) %>%
  map("Mean") %>%
  as.vector %>%
  map(max) %>%
  print
```

```
## [[1]]
## [1] 0.3037799
##
## [[2]]
## [1] 0.1508972
##
## [[3]]
## [1] 0.1037533
```

Mapping over multiple inputs `purrr::map2` takes two different input parameters and a function that takes 2 arguments.

```
counts <- sample(1:100, 3)
percent <- rnorm(3, mean=0.5, sd=0.1)
map2(counts, percent, ~ paste0(.x, " (", round(.y * 100, digits=1), "%)"))
```

```
## [[1]]
## [1] "95 (54.5%)"
##
```

```
## [[2]]
## [1] "38 (44.7%)"
##
## [[3]]
## [1] "55 (45.3%)"
```

In base R we have `mapply`, which first takes a function and then it's inputs.

```
mapply(function(x,y) paste0(x, " (", round(y * 100, digits=1), "%)"), counts, percent)
```

```
## [1] "95 (54.5%)" "38 (44.7%)" "55 (45.3%)"
```

Both `purrr` and base R have their pros and cons, and it is entirely personal preference which you prefer.

Rounding in an ADLB table

Consider an example of a possible ADLB dataset. (Only the first six rows are shown)

USUBJID	PARAMCD	PARCAT	AVAL	AVALC	AVISIT
101001-101101	LB00185	Cat 2	3.008100	3.0081	Day 1
101001-101101	LB00185	Cat 2	2.952000	2.952	Day 2
101001-101101	LB00185	Cat 2	3.799200	3.7992	Day 3
101001-101101	LB00185	Cat 2	3.398850	3.39885	Day 4
101001-101101	LB00185	Cat 2	1.062292	1.062292	Day 5
101001-101101	LB00192	Cat 1	8.304605	8.304605	Day 1

In this scenario we want to correctly work out how many decimal places we should be displaying per `PARAMCD` in our output table. In this case we are going to display to the mean number of decimal places.

The general method for this in SAS is to create a separate dataset to calculate the values, and then merge a new variable onto the existing dataset and round using this new variable.

In R we can create a higher order function. Which takes the input of our grouped data `PARAMCD` and returns a function that rounds our values to the correct DP.

```
round_by_paramcd <- function(group) {
  # Calculate the number of DPS:
  # Get all vales after & incl. the decimal point (minus 1 for the decimal place itself)
  str_ext <- nchar(str_extract(group, "\\..*")) - 1
  # Replace NA values (no decimal) with zeros
  str_ext[is.na(str_ext)] <- 0
  # Work out the mean and round to nearest values
  dps <- round(mean(str_ext), 0)

  # Our anonymous function to round to the correct DP
  function(x) {
    round(x, digits = dps) %>%
      format(., nsmall = dps) #ensure the correct number of DPS are shown
  }
}
```

We can then call our function with `AVALC` being used to call our function, and then calling the returning function with our mean value of `AVAL`

```
ADLB %>%
  group_by(PARAMCD) %>%
  summarise(mean = round_by_paramcd(AVALC)(mean(AVAL)))
```

PARAMCD	mean
LB00185	1.82
LB00192	8.02757
LB00283	3.9978

We can verify this has rounded as we expect with the following. Bear in mind that R rounds half to even as per the IEEE 754 standard.

```
ADLB %>%
  group_by(PARAMCD) %>%
  mutate(dps=nchar(str_extract(AVALC, "\\..*")) - 1,
         dps=if_else(is.na(dps), 0, dps)) %>%
  summarise(mean(dps), round(mean(dps)))
```

PARAMCD	mean(dps)	round(mean(dps))
LB00185	2.171875	2
LB00192	5.161765	5
LB00283	3.808823	4

An additional benefit of the function is that we can adjust it to any grouping we desire without any further changes to the function.

```
ADLB %>%
  group_by(PARCAT) %>%
  summarise(mean = round_by_paramcd(AVALC)(mean(AVAL))) #Apply our function
```

PARCAT	mean
Cat 1	6.0127
Cat 2	1.82

We have abstracted the problem to a simple solution which can be reused and transferred between projects.

Apply attributes to columns in a data frame

`walk` is a unique function in the `purrr` package with no equivalent in base R. `walk` is like `map` but the input is returned unchanged. An example of this is the following

```
x <- 1:5
walk(x, ~ . + 1) %>% print
```

```
## [1] 1 2 3 4 5
```

```
map_dbl(x, ~ . + 1) %>% print
```

```
## [1] 2 3 4 5 6
```

However, the *side effects* of the function can still occur. Consider a different example, using the global assignment operator `<<-`.

```
walk(x, function(x) { attr(x, "PHUSE") <<- "Hi EU Connect!" })
attributes(x)
```

```
## $PHUSE
## [1] "Hi EU Connect!"
```

We can use the above to add labels to our datasets, such as the ADLB dataset from before.

```
labels <- c(USUBJID="Unique Subject Identifier",
            PARAMCD="Parameter Code",
            PARCAT="Parameter Category",
            AVAL="Value",
            AVALC="Value (Character)",
            AVISIT="Analysis Visit")

walk2(names(labels),
      labels,
      function(x,y) { attr(ADLB[[x]], "label") <<- y })
head(ADLB)
```

USUBJID Unique Subject Identifier	PARAMCD Parameter Code	PARCAT Parameter Category	AVAL Value	AVALC Value (Character)	AVISIT Analysis Visit
101001-101101	LB00185	Cat 2	3.008100	3.0081	Day 1
101001-101101	LB00185	Cat 2	2.952000	2.952	Day 2
101001-101101	LB00185	Cat 2	3.799200	3.7992	Day 3
101001-101101	LB00185	Cat 2	3.398850	3.39885	Day 4
101001-101101	LB00185	Cat 2	1.062292	1.062292	Day 5
101001-101101	LB00192	Cat 1	8.304605	8.304605	Day 1

References

Abelson, Harold, and Gerald Jay Sussman. 1996. *Structure and Interpretation of Computer Programs*. The MIT Press.

R Core Team. 2019. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org.

Wickham, Hadley. 2019. *Advanced r*. CRC press.

Wickham, Hadley, and Garrett Grolemund. 2016. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. " O'Reilly Media, Inc.".