Functional Programming

Biostatistics 140.776

What is Functional Programming?

Functional programming concentrates on four constructs:

1. Data (numbers, strings, etc)
2. Variables (function arguments)
3. Functions
4. Function Applications (evaluating functions given arguments and/or data)

What is Functional Programming?

By now you're used to treating variables inside of functions as data:

- numbers and strings
- data structures like lists and vectors.

With functional programming you can also

- provide a **function** as an argument to another function
- return a function as the result of a function

The `sapply()` function is a an example of supplying a function as an argument to another function

Simple Example

The following function returns a function as its “result”.

```
adder_maker <- function(n) {
        function(x){
                n + x
        }
}
```

This function “constructs” functions based on its argument `n`.

Simple Example

```
## Make a function that adds 3 to its argument
add3 <- adder_maker(3)
add3(5)
```

```
[1] 8
```

```
## Make a function that adds 2 to its argument
add2 <- adder_maker(2)
add2(5)
```

```
[1] 7
```

Simple Example

A few notes on this example:

- This example works because each of the `add` functions "knows" its value of `n`
- Lexical scoping rules mean that both `add2()` and `add3()` look up the value of `n` in the environment where they were defined
- Each function was defined inside the `adder_maker()` function which supplied the value of `n` via the user

Functional Programming

The functional programming approach essentially allows you to **write code** based on **data** supplied by the user!

1. Usual model: Data –> Code –> "Results"
2. Functional model: Data –> Code –> Code (!)

It's a very powerful technique that can allow you to develop so-called "intelligent" systems

The `purrr` Package

- There are groups of functions that are essential for functional programming: map, reduce, filter, compose, search
- In most cases they take a function and a data structure as arguments, and that function is applied to that data structure in some way.
- The `purrr` package contains many of these functions
- Functional programming is concerned mostly with lists and vectors (for data structures)

Map

A key function in the `purrr` package is the `map()` function, which has two arguments:

- `.x` A data structure like a list or vector
- `.f` A function to be applied, a formula (converted into a function), or an atomic vector

`map()` always returns a list, regardless of the input. It works much like `lapply()`.

Map

```
library(purrr)
num2word <- function(x) {
        words <- c("one", "two", "three", "four", "five")
        words[x]
}
map(3:1, num2word)
```

```
[[1]]
[1] "three"

[[2]]
[1] "two"

[[3]]
[1] "one"
```

Map with Simplification

```r
map_chr(5:1, num2word)
```

```
[1] "five"  "four"  "three" "two"   "one"
```

Map

Often we want to map a function to an object (list/vector) and then extract an element of each of the return values.
This code splits the MIE data into separate rooms, fits a linear model to each subset, and then extracts the coefficients.

```
results <- split(airquality, airquality$Month) %>%
        map(~ lm(Ozone ~ Temp, data = .x)) %>%
        map("coefficients")
```

Map

```
results[1:3]
```

```
$`5`
(Intercept)        Temp
-102.159308    1.884808

$`6`
(Intercept)        Temp
 -91.990958    1.552441

$`7`
(Intercept)        Temp
-372.920837    5.150363
```

Map

Using `map()` with a formula argument "auto-generates" an anonymous function, which is more compact than using something like `lapply()`.

```
split(airquality, airquality$Month) %>%
        map(~ lm(Ozone ~ Temp, data = .x))
```

is equivalent to

```
lapply(split(airquality, airquality$Month),
       function(.x) {
               lm(Ozone ~ Temp, data = .x)
       })
```

The version using `map()` is more readable, modular, and highlights what is happening in the right sequence.

Conditional Map

The `map_if()` function takes a **predicate** and then a function for application. A predicate is a function that returns TRUE or FALSE given some input.

```
## A predicate function
is_even <- function(x) {
        x %% 2 == 0
}
square <- function(y){
        y^2
}
map_if(1:5, is_even, square) %>% unlist
```

```
[1]  1  4  3 16  5
```

If the predicate evalutes to FALSE, the argument just passes through.

Reduce

List or vector reduction is done with the `reduce()` function. It is a kind of looping that

1. Combines the first element of a vector with the second element of a vector;
2. then that combined result is combined with the third element of the vector;
3. and so on until the end of the vector is reached.

The function to be applied should take at least **two** arguments.

Reduce

Often, after mapping a function to a list/vector, you will want to combine the results at the end.

```
map_if(1:5, is_even, square) %>%
        reduce(function(x, y) {
                x + y
        })
```

```
[1] 29
```

This is overkill since the result is just a simple list; we could have just used `sum()`.
But what if the `map()` function returns a more complicated object?

Reduce

```
split(airquality, airquality$Month) %>%
        map(~ lm(Ozone ~ Temp, data = .x)) %>%
        map("coefficients") %>%
        reduce(function(x, y) {
                ## 'x' and 'y' are numeric vectors
                cbind(x, y)
        })
```

```
                       x          y           y           y
(Intercept) -102.159308 -91.990958 -372.920837 -238.861312
Temp           1.884808   1.552441    5.150363    3.559044
```

Search

You can search for specific elements of a vector using the

- `contains()`: will return `TRUE` if a specified element is present in a vector, otherwise it returns `FALSE`
- `detect()`: takes a vector and a predicate function as arguments and it returns the first element of the vector for which the predicate function returns ‘TRUE

Search

```r
purrr::contains(letters, "a")
```

```
[1] TRUE
```

```r
purrr::contains(letters, "A")
```

```
[1] FALSE
```

Search

```r
detect(20:40, function(x){
  x > 22 && x %% 2 == 0
})
```

```
[1] 24
```

Filter

The various filter functions are

- `keep()`: Keep elements that satisfy a predicate
- `discard()`: Remove elements that satisfy a predicate
- `every()`: returns TRUE only if every element in the vector satisfies the predicate function
- `some()`: returns TRUE if at least one element in the vector satisfies the predicate function

Filtering

```
keep(1:20, function(x){
  x %% 2 == 0
})
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

```
discard(1:20, function(x){
  x %% 2 == 0
})
```

```
 [1]  1  3  5  7  9 11 13 15 17 19
```

Filtering

```r
every(1:20, function(x){
  x %% 2 == 0
})
```

```
[1] FALSE
```

```r
some(1:20, function(x){
  x %% 2 == 0
})
```

```
[1] TRUE
```

Compose

The `compose()` function take any number of functions and combines them into a single function.

```
nlevels <- compose(length, unique)
nlevels(c("a","a", "a", "b", "b"))
```

```
[1] 2
```

This is equivalent to

```
nlevels <- function(x) {
        length(unique(x))
}
```

The evaluation of functions is done from right to left.

Compose

You can also do more unusual things like

```
not_null <- compose("!", is.null)
not_null(NULL)
```

```
[1] FALSE
```

```
not_null(4)
```

```
[1] TRUE
```

Compose

You can also build upon little building blocks

```
add1 <- function(x) {
        x + 1
}
add3 <- compose(add1, add1, add1)
add3(8)
```

```
[1] 11
```

Compose

When fitting linear models, I do this a lot:

```
summary_lm <- compose(summary, lm)
summary_lm(Ozone ~ Temp, data = airquality)
```

```

Call:
last(formula = ..1, data = ..2)

Residuals:
    Min      1Q  Median      3Q     Max
-40.729 -17.409  -0.587  11.306 118.271

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -146.9955    18.2872  -8.038 9.37e-13
Temp           2.4287     0.2331  10.418  < 2e-16

Residual standard error: 23.71 on 114 degrees of freedom
```

Compose

Or even

```
coef_lm <- compose(coef, lm)
coef_lm(Ozone ~ Temp, data = airquality)
```

```
(Intercept)        Temp
-146.995491    2.428703
```

Summary

- Functional programming treats functions as “first class” citizens, along with data, and parameters
- Building functions that adapt to data is a key element
- Map, reduce, filter, search, and compose are key concepts
- Often can be a clearer way to express code (if not more succinct or faster)
- Functional programming often extends easily to parallel programming