Functional programming

450X

Stanford University

Department of Political Science

Toby Nowacki

Overview

1. Why functions?
2. Common pitfalls
3. Functionals
4. Function factories
5. Recursion

Why functions?

- A function is a mapping from some inputs $\mathbf{X}$ to some outputs $\mathbf{Y}$.
- Whenever we carry out the same process more than once, a function is strongly recommended
- Much more convenient for both tractability and debugging
- Allow for decomposition of complex problems into smaller pieces

Basic function architecture

```
foo ← function(x, y){
    return(x + y)
}
```

Common pitfalls (1)

Can you spot the problem?

```r
foo <- function(x, y){
    return(x + y)
}

item1 <- 3
item2 <- "five"
```

```r
foo(item1, item2)
```

Solution (1)

```r
foo ← function(x, y){
    stopifnot(is.numeric(x), is.numeric(y))
    return(x + y)
}
```

```
foo(item1, item2)
# Error in foo(item1, item2) : is.numeric(y) is not TRUE
# Calls: <Anonymous> ... withCallingHandlers → withVisibl
```

- An alternative is using `tryCatch()`.

Common pitfall (2)

What's wrong here?

```
bar ← function(x, y, z){
    out ← x + y
    return(out)
    out_two ← out + z
    return(out_two)
}
```

Common pitfall (2)

What's wrong here?

```r
bar <- function(x, y, z){
    out <- x + y
    return(out)
    out_two <- out + z
    return(out_two)
}

bar(2, 4, 6)
```

```
## [1] 6
```

Solution (2)

```r
bar <- function(x, y, z){
    out <- x + y
    cat(paste0("Intermediate output: ", out))
    out_two <- out + z
    return(out_two)
}

bar(2, 4, 6)
```

```
## Intermediate output: 6

## [1] 12
```

Functionals

- Functions can take *other functions* as arguments!
- we've seen this before in the form of `lapply` or `map`:

```r
vec <- 2:6
map_dbl(vec, sqrt)
```

```
## [1] 1.414214 1.732051 2.000000 2.236068 2.449490
```

- Other functions that rely on functionals are, for example, `apply`, `optimize`, `integrate`

Functionals (cont'd)

- You can write your own functions with functionals:

```r
print_summary ← function(data, fn){
    out ← fn(data)
    return(paste0("Statistic: ", out))
}
print_summary(c(2, 4, 4), mean)
```

```
## [1] "Statistic: 3.33333333333333"
```

```r
print_summary(c(2, 4, 4), max)
```

```
## [1] "Statistic: 4"
```

Functionals (cont'd)

- But what about this?

```
blob <- c(2, 4, 4, NA)
print_summary(blob, mean)
```

```
## [1] "Statistic: NA"
```

- Can't pass additional arguments to `mean`:

  ```
  print_summary(blob, mean(na.rm = FALSE))
  ```

Functionals (cont'd)

- Fortunately, there is a shortcut!
- `...` lets us pass on whatever else is specified as an input argument.

```r
print_summary <- function(x, f, ...){
    return(f(x, ...))
}
print_summary(blob, mean, na.rm = TRUE)
```

```
## [1] 3.333333
```

Functionals (cont'd)

- Selecting columns in dataframes is a little bit trickier.

```r
df <- tibble(name = c("A", "B", "C"),
             value = c(30, 16, 45))

col_summary <- function(dataframe, col_name, f,
    get_col <- dataframe %>% dplyr::select(col_na
    return(f(get_col, ...))
}
```

```r
col_summary(df, value, mean)
# Error in .f(.x[[i]], ...) : object 'value' not found
```

Functionals (cont'd)

- Have to rely on something called `tidyeval`
- Look up quotations and quasi-quotations!

```r
col_summary <- function(dataframe, col_name, f,
    col_name <- enquo(col_name)
    get_col <- dataframe %>%
        summarise(out = f(!!col_name, ...))
    return(get_col)
}
col_summary(df, value, mean, na.rm = TRUE)
```

```
## # A tibble: 1 x 1
##     out
##   <dbl>
## 1  30.3
```

Function factories

- Functions can also produce *other* functions as output!
- These things are sometimes called `function factories`.

```r
factory <- function(x, y){
    fm <- paste0(y, " ~ ", x)
    function(d){
        lm(formula = fm, data = d)$coef
    }
}
```

Function factories (cont'd)

```r
car_reg ← factory("mpg", "hp")
car_reg(mtcars)
```

```
## (Intercept)         mpg
##   324.082314   -8.829731
```

Function factories (cont'd)

```
car_reg2 <- factory("cyl", "wt")
car_reg2(mtcars)
```

```
## (Intercept)         cyl 
##   0.5646195   0.4287080 
```

Function factories (cont'd)

- Will be very useful when doing bootstrapping or MLE estimation

Recursion

- Factorial example:

$$n! = n * (n-1) * (n-2) * \ldots * 1$$

- Use the property of recursion to make the function to refer to itself.

Recursion (cont'd)

- What's wrong with the definition as below?

```r
factorial_fn <- function(n){
    return(n * factorial_fn(n-1))
}
```

Recursion (cont'd)

- Let's fix it.

```
factorial_fn ← function(n){
    if(n ⩽ 1){
        return(1)
    }
    else{
        return(n * factorial_fn(n-1))
    }
}
```

Recursion (cont'd)

```r
factorial_fn(5)
```

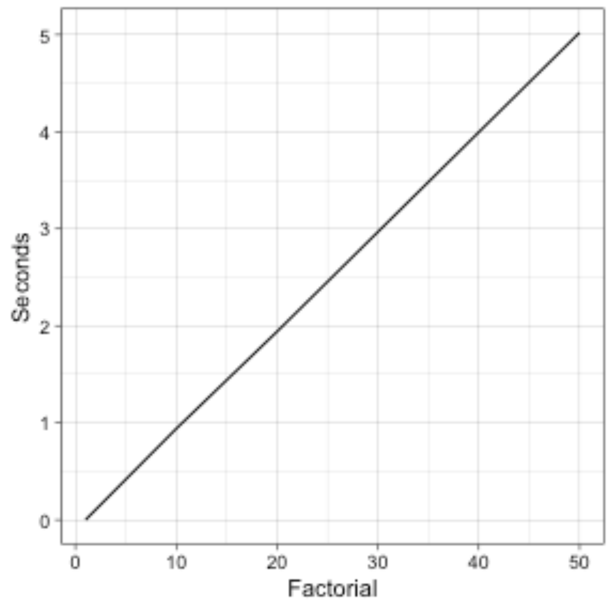
```
## [1] 120
```

```r
factorial_fn(4)
```

```
## [1] 24
```

Problems with recursion

- Not always the most efficient implementation...



Further applications

- Fibonacci sequence $x_n = x_{n-1} + x_{n+2}$
- Collatz conjecture (Syracuse Problem)
- Sorting, searching, merging algorithms...

Conclusion

- More hands-on programming: what are the most efficient ways to solve a problem?
- Functions are the bread-and-butter of intermediate and advanced programming
- Highly recommended for replicability, tractability, and time saving.
- Still, much more out there... (e.g., basic search algorithms)

Next week

- Parallel programming
- Server-side scripts and working on the cluster