Statistical Programming and Open Science Methods

Functional versus object-oriented programming

Joachim Gassen
Humboldt-Universität zu Berlin

September 02, 2022

SFB/Transregio 266
ACCOUNTING FOR TRANSPARENCY

Time table October 10

When?	What?
09:00	Welcome and Introduction
09:30	The development environment and project organization
10:30	Coffee
11:00	Using Git and Github
12:30	Lunch
14:00	Statistical programming languages: An overview
15:30	Coffee
16:00	Functional versus object-oriented programming
19:30	Pizza at Due Forni, Schönhauser Allee 12

Functional programming versus scripting

- Many statistical programming languages (EViews, SAS, Stata, R to some extent) are in essence scripting languages.
- Scripts are closely connected to imperative programming ("Shut up and do what I tell you!")
- Scripts are hard to read, tend to become inefficient, and are hard to reuse
- "If you copy + paste your (own) code a lot, you are a bad programmer"

The key idea of functional programming

- Functional programming is declarative in nature: Your functions describe what to do. The implementation is hidden from the user.
- A function takes arguments, processes them and returns results
- A *pure function* is a function where
  - the result of the function depends only on its arguments and
  - that generates no *side effects*
- Pure functions are *referentially transparent*, meaning that they can be replaced with their return value without changing the program
- In real-life coding, many functions are not referentially transparent. This makes writing code easier and reading code harder

Functions in R

- Functions have three components:
  - `formals()`: The arguments that you call the function with
  - `body()`: The code that the function executes
  - `environment()`: The place where the function can look for objects
- Functions are objects, just like about anything else in R
- Internally, they are called `closures`. Knowing this can be helpful to decipher error messages!

Chaining functions in R: Intermediate objects

Readable but tedious

```
df <- read_csv("data/sub.csv")
df <- select(df, cik, name)
df <- distinct(df)
count_sec_reg <- nrow(df)
sprintf("There are %d registrants", count_sec_reg)
```

Chaining functions in R: Nesting

Concise but a pain in the eye

```
sprintf(
  "There are %d registrants",
  nrow(distinct(select(read_csv("data/sub.csv"), cik, name)))
)
```

Chaining functions in R: Piping

The tidy way (read %>% as "and then") but harder to debug

```
read_csv("data/sub.csv") %>%
  select(cik, name) %>%
  distinct() %>%
  nrow() -> count_sec_reg

sprintf("There are %d registrants", count_sec_reg)
```

Scoping I

What does this code snippet return?

```
x <- 10

my_func <- function() {
  x <- 20
  x
}

c(my_func(), x)
```

Scoping I

What does this code snippet return?

```r
x <- 10

my_func <- function() {
  x <- 20
  x
}

c(my_func(), x)
```

```
## [1] 20 10
```

Scoping II

What does this code snippet return?

```
x <- 10
y <- 5

my_func <- function() {
  x <- 20
  x*y
}

my_func()
```

Scoping II

What does this code snippet return?

```
x <- 10
y <- 5

my_func <- function() {
  x <- 20
  x*y
}

my_func()
```

```
## [1] 100
```

Scoping III

What does this code snippet return?

```
my_second_func <- function(x) {
  y <- x
}

my_func <- function(x) {
  x*y
}

my_second_func(5)
my_func(10)
```

Scoping III

What does this code snippet return?

```r
my_second_func <- function(x) {
  y <- x
}

my_func <- function(x) {
  x*y
}

my_second_func(5)
my_func(10)
```

```
## Error in my_func(10): object 'y' not found
```

Scoping IV

What does this code snippet return?

```r
my_second_func <- function(x) {
  y <<- x
}

my_func <- function(x) {
  x*y
}

my_second_func(5)
my_func(10)
```

Scoping IV

What does this code snippet return?

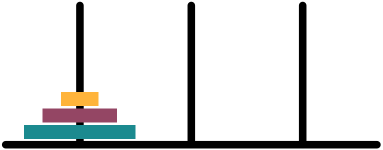
```
my_second_func <- function(x) {
  y <<- x
}

my_func <- function(x) {
  x*y
}

my_second_func(5)
my_func(10)
```

```
## [1] 50
```

For extra credit

Which of these functions are pure, which are not? Why?

Recursions: Functions can be very helpful



```r
tower <- function(n, from_peg, to_peg, aux_peg) {
  if(n == 0) return(invisible())
  tower(n - 1, from_peg, aux_peg, to_peg)
  message(sprintf("Moving piece %d from %s to %s ...",
                  n, from_peg, to_peg), appendLF = FALSE)
  tower(n - 1, aux_peg, to_peg, from_peg)
}
tower(3, 'F', 'T', 'A')
```

Recursions: Functions can be very helpful

```
tower <- function(n, from_peg, to_peg, aux_peg) {
  if(n == 0) return(invisible())
  tower(n - 1, from_peg, aux_peg, to_peg)
  message(sprintf("Moving piece %d from %s to %s ...",
                  n, from_peg, to_peg))
  tower(n - 1, aux_peg, to_peg, from_peg)
}
tower(3, 'F', 'T', 'A')
```

```
## Moving piece 1 from F to T ...

## Moving piece 2 from F to A ...

## Moving piece 1 from T to A ...

## Moving piece 3 from F to T ...

## Moving piece 1 from A to F ...

## Moving piece 2 from A to T ...

## Moving piece 1 from F to T ...
```

See https://www.youtube.com/watch?v=YstLjLCGmgg for animation

Object oriented programming

- Much more common in Python that in R, object oriented programming encapsulates data and functions (aka as *methods* in the OOP world) in *classes*
- Methods can be *overloaded* by *inheriting* classes
- Tends to make code more consistent and easier to maintain/extend
- Makes it easier for code to modify data (makes objects more *mutable*), something that people in statistical programming are generally not very fond off
- How does this look like: Let's have a quick look at a last toy example
  - `code/show_fs_oop.py` versus
  - `code/show_fs_fp.R`