DSCI351-351m-451: Class 8a Functional vs Object Oriented Programming

2208-351-351m-451-10a-p-FunctionalObjectOriented

Paul W. Leu

19 October, 2022

Contents

8.0.1.1 Class Readings, Assignments, Syllabus Topics

8.0.1.1.1 Reading, Lab Exercises, SemProjects

- Readings:
  - For today:
  - For next class:
- Laboratory Exercises:
  - LE 4: due today
  - LE 5: Thursday, 11/10
- Office Hours: (Class Canvas Calendar for Zoom Link)
  - Mondays @ 4:30 PM to 5:30 PM
  - Wednesdays @ 4:30 PM to 5:30 PM

  - **Office Hours are on Zoom, and recorded**
- Semester Projects
  - DSCI 451 Students Biweekly Update 4 Due this Friday
  - DSCI 451 Students
    - Next Report Out #2 is Due Friday October 28th
  - All DSCI 351/351M/451 Students:
    - Final: Monday 12/13/2021, 12:00PM - 3:00PM, Nord 356 or remote

Day:Date	**Foundation**	**Practicum**	**Reading**	**Due**
w01a:Tu:8/30/22	ODS Tool Chain	R, Rstudio, Git		
w01b:Th:9/1/22	Setup ODS Tool Chain	Bash, Git, Slack, Agile	PRP4-33	LE1
w02a:Tu:9/6/22	Bash-Git-Knuth-Lit.Prog.	RIntroR	PRP35-64	
w02b:Th:9/8/22	What is Data Science	OIS:Intro2R	OIS1,2	
w02Pr:Fr:9/9/22			PRP65-93	**451 Update**
w03a:Tu:9/13/22	Data Intro	Data Analytic Style	PRP94-116	LE2 **LE1 Du**
w03b:Th:9/15/22	Rand. Var. Normal Dist.	Git, Rmds, Loops	OIS4	
w04a:Tu:9/20/22	Tidy Check Explore	Tidy GapMinder	EDA1-31	
w04b:Th:9/22/22	Inference, DSCI Process	Other Distrib. 7 ways	R4DS1-3	LE3 **LE2 Du**
w04Pr:Fr:9/23/22			EDA32-58	**451 Update**
w05a:Tu:9/27/22	OIS4 Rand. Var.	EDA of PET Degr.	OIS5	
w05b:Th:9/29/22	OIS5 Found. of Infer.	Multivar Corr. Plot	R4DS4-6	
w05Pr:Fr:9/30/22				**451 RepOu**
w06a:Tu:10/4/22	Pred., Algorithm, Model		R4DS7-8	
w06b:Th:10/6/22	Summ. Stats & Vis.	Anscombe's Quartets	R4DS9-16	LE4 **LE3 Du**
w06Pr:Fr:10/7/22				**451 Update**
w07a:Tu:10/11/22	Midterm Rev. Tidy Data	Correl Plots Summ Stats	OIS6.1-2	**PeerRv1 Du**
w07b:Th:10/13/22	HypoTest, Infer. Recap	Penguin EDA, Sampling		
w08a:Tu:10/18/22	**MIDTERM**	**EXAM**		
w08b:Th:10/20/22	Programming & Coding	Code Packaging		**LE4 Due**
w08Pr:Fr:10/21/22				**451 Update**
Tu:10/24,25	**CWRU**	**FALL BREAK**	R4DS17-21	
w09b:Th:10/27/22	Cat. Inf. 1 & 2 propor.	Indep. Test,2-way tables	OIS6.3-4	LE5
w09Pr:Fr:10/28/22				**451 RepOu**
w10a:Tu:11/1/22	Goodness of Fit, χ^2 test	t-tests 1&2 means	OIS7.1-4	
w10b:Th:11/3/22	Num. Infer, Cont. Tables	Stat. Power		
w10Pr:Fr:11/4/22				**451 Update**
w11a:Tu:11/8/22	Sample & Effect Size	Stat. Power GGmap	OIS8	**PeerRv2 Du**
w11b:Th:11/10/22	Inf. 4 Regr, Test & Train	Curse of Dimen.	ISLR1,2.1,2	LE6 **LE5 Du**
w12a:Tu:11/15/22	Lin. Regr. Part 1	Residuals	OIS9	
w12b:Th:11/17/22	Lin. Regr. Part 2	Regr. Diagnostics		
w12Pr:Fr:11/18/22				**451 Update**
w13a:Tu:11/22/22	Mult. Lin. Regr.	Var. & Mod. Selec.,	ISLR3.1	LE7 **LE6 du**
w13b:Th:11/24/22	Log. Regr.	GIS Trends	ISLR3.2	
w13Pr:Fr:11/25/22				**451 RepOu**
w14a:Tu:11/23/22	Classificat., Sup. Lrning	Caret, Broom 4 modeling	ISLR4.1-3	
Th,Fr:11/24,25	**THANKSGIVIING**	**Vacation**		
w15a:Tu:11/29/22		Clustering		**PeerRv3 Du**
w15b:Th:12/1/22	Big Data Analytics	Dist. Comp., Hadoop		
w15SPr:Fr:12/2/22		Read Article by	Mirletz,2015	
w16a:Tu:12/6/22	Final Exam Review			
w15b:Th:12/8/22				**LE7 due**
Friday 12/12	**SemProj**	**Final Report**		**SemProj4 d**
Monday 12/19	**FINAL EXAM**	**12:00-3:00pm**	Nord 356	or remote

8.0.1.2 Syllabus

### Functional vs Object Oriented Programming

Two programming paradigms

- Functional programming is a paradigm based on writing functions

- Object oriented programming is a paradigm based around `objects`, which can be data and code
- R is a functional language.

- OOP is more challenging in R because
  - there are multiple OOP systems (S3, R6, S4)

Typically in R, you use functional programming, where you solve complex problems by

- decomposing them into simple functions, not objects.

8.1 Functional programming

Decompose a big problem into smaller pieces, then solve each piece with a function of combination of functions.

8.1.1 Functions

Functions are composed of

- arguments or `formals()`, that control how you call the function
- body or `body()`, the code inside the function
- environment or `environment()`, the data structure that determines how the function finds values associated with the names.

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

formals(Sum)
```

```
## $x
##
##
## $y
```

```
body(Sum)
```

```
## {
##     return(x + y)
## }
```

```
environment(Sum)
```

```
## <environment: R_GlobalEnv>
```

R also has many primitive functions, which call C code directly.

Primitive functions are written in R so their `formals()`, `body()`, and `environment()` are all NULL:

```
typeof(sum)
```

```
## [1] "builtin"
```

```
formals(sum)
```

```
## NULL
```

```
body(sum)
```

```
## NULL
```

```
environment(sum)
```

```
## NULL
```

R functions are objects and often called *first-class functions.* They can, just as any other object, be

- bound to names,
- passed as arguments, and
- returned from other functions.

When you name a function, use a "command" verb in CamelCase. Keep your functions short (30 lines or less), so that they can be combined in a modular manner.

You can use *anonymouse functions* if you think you will only use a function once and do not want to give a name.

```
sapply(mtcars, function(x) length(unique(x)))
```

```
##  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb 
##   25    3   27   22   22   29   30    2    2    3    6
```

In R, functions are called *closures.* This is because they enclose their environments.

```
typeof(Sum)
```

```
## [1] "closure"
```

```
body(Sum)
```

```
## {
##     return(x + y)
## }
```

```
y <- 1
f <- function(x) x + y
f(3)
```

```
## [1] 4
```

8.1.2 Functionals

A *functional* is a function that takes a function as an input and returns a vector as an output.

`lapply()`, `sapply()`, `apply()` are functionals.
Mathematical functions like `integrate()` are functionals.

```
sapply(mtcars, typeof)
```

```
##      mpg      cyl     disp       hp     drat       wt     qsec       vs 
## "double" "double" "double" "double" "double" "double" "double" "double" 
##       am     gear     carb 
## "double" "double" "double"
```

```
integrate(function(x) sin(x), 0, pi)
```

```
## 2 with absolute error < 2.2e-14
```

The most important functional is `map()` which is included as part of tidyverse. It takes - a vector and - a function and returns - a list

`map(1:3, f)` is equivalent to `list(f(1), f(2), f(3))`

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.3.6      v purrr   0.3.5
## v tibble  3.1.8      v dplyr   1.0.10
## v tidyr   1.2.1      v stringr 1.4.1
## v readr   2.1.3      v forcats 0.5.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
triple <- function(x) x * 3
map(1:3, triple)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 9
```

There are variants of `map()`: `map_lgl()`, `map_int()`, `map_dbl()`, `map_chr()`, and `map_dfr()`.
Each returns an atomic vector of a different type: logical, integer, double, character, and dataframe, respectively.

```
map_chr(mtcars, typeof)
```

```
##      mpg      cyl     disp       hp     drat       wt     qsec       vs
## "double" "double" "double" "double" "double" "double" "double" "double"
##       am     gear     carb
## "double" "double" "double"
```

```
mtcars_by_cyl <- split(mtcars, mtcars$cyl)
slopes <- double(length(mtcars_by_cyl))
intercepts <- double(length(mtcars_by_cyl))
for (i in seq_along(mtcars_by_cyl)) {
  model <- lm(mpg ~ wt, data = mtcars_by_cyl[[i]])
  slopes[[i]] <- coef(model)[[2]]
  intercepts[[i]] <- coef(model)[[1]]
}
df_model <- as.data.frame(t(rbind(intercepts, slopes)))


df_model2 <- mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .x)) %>%
  map_dfr(~ as.data.frame(t(as.matrix(coef(.))))) %>%
  rename(intercepts = `(Intercept)`, slopes = wt)
```

8.2 Object Oriented Programming

Main reason to use OOP is *polymorphism*.
Polymorphism means that a develop can consider a function's interface seprately from the implementation.

This is related to *encapsulation* where the user doesn't have to worry about the details of an object. This facilitates code refactoring. How data is represented internally can be changed without worrying about how external code interacts with this object.

Polymorphism is what allows `summary()` to produce different outputs for numeric and factor variables.

```
summary(diamonds$carat)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##  0.2000  0.4000  0.7000  0.7979  1.0400  5.0100
```

```
summary(diamonds$cut)
```

```
##      Fair      Good Very Good   Premium     Ideal 
##      1610      4906     12082     13791     21551
```

The type of object is its *class* and an implementation for a specific class is called a *method*.

The class defines the *fields*, the data possessed by every instance of that class. Classes are organised in a hierarchy so that if a method does not exist for one class, its parent's method is used, and the child is said to inherit behaviour.

In encapsulated OOP, methods belong to objects or classes, and method calls typically look like `object.method(arg1, arg2)`. This is called encapsulated because the object encapsulates both data (with fields) and behaviour (with methods), and is the paradigm found in most popular languages.

```
setClass("Student",
  slots = c(
    name = "character",
    IDnumber = "numeric",
    email = "character",
    team = "character"
  )
)

john <- new("Student", name = "John Smith", email = "john.smith@pitt.edu", IDnumber = 1234567, team = "
john@name

setMethod("send_email", "Student", function(x) {
  # code to send email
}
```

```
## Error: <text>:17:0: unexpected end of input
## 15: }
## 16: 
##    ^
```

8.2.0.1 Links