P07 – Objektorientierte Programmierung

24. Mai 2021

Contents

Hinweise zur Abgabe:

Erstelle pro Aufgabe eine R-Code-Datei (in diesem Fall 3 Dateien) und benenne diese nach dem Schema `P<Woche>-<Aufgabe>.R` also hier `P07-1.R`, `P07-2.R` und `P07-3.R`. Schreibe den Code zur Lösung einer Aufgabe in die jeweilige Datei.

Es ist erlaubt (aber nicht verpflichtend) zu zweit abzugeben. Abgaben in Gruppen von drei oder mehr Personen sind nicht erlaubt. Diese Gruppierung gilt nur für die Abgabe der Programmierprobleme.

Bei Abgaben zu zweit gibt nur eine der beiden Personen ab. Dabei müssen in **jeder** abgegebenen Datei in der **ersten Zeile** als Kommentar **beide** Namen stehen also zB

```
# Ada Lovelace, Charles Babbage

1+1
# ...
```

Die Abgabe der einzelnen Dateien (kein Archiv wie `.zip`) erfolgt über Moodle im Element namens `P07`. Die Abgabe muss bis spätestens Sonntag, 30. Mai 2021, 23:59 erfolgen.

Funktionsoperator (10 Punkte)

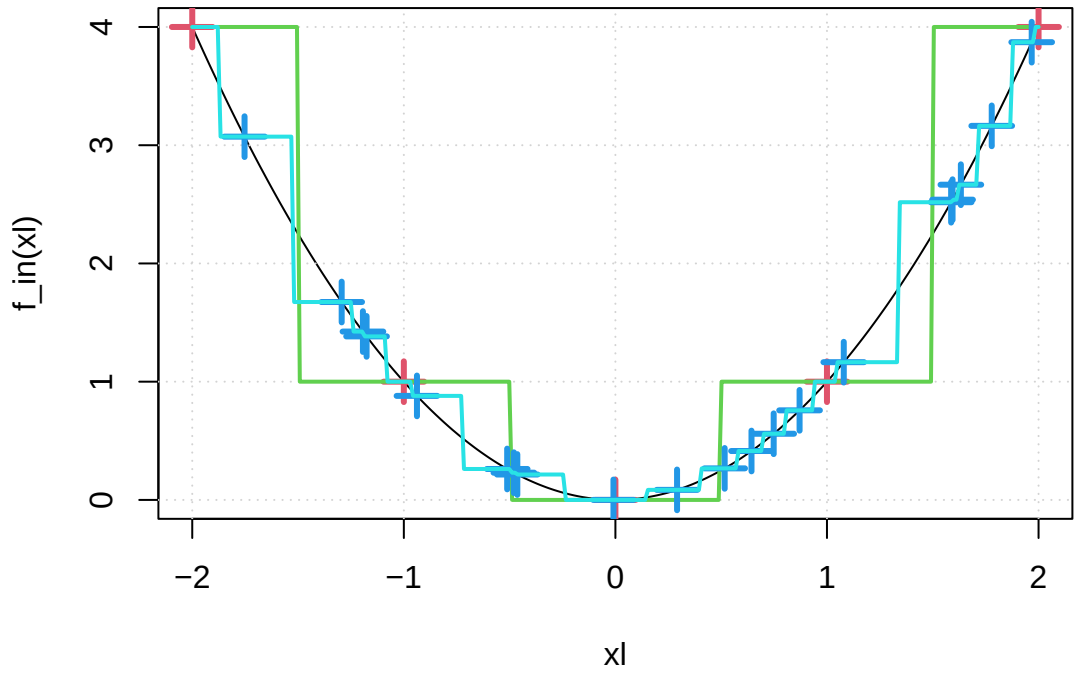
Erzeuge einen Funktionsoperator `mem_near()`, der eine Funktion `f_in()` entgegen nimmt und eine neue Funktion `f_out()` ausgibt. Wir gehen davon aus, dass `f_in()` eine vektorisierte Version einer Funktion $\tilde{f}\colon \mathbb{R} \to \mathbb{R}$ ist, dh $f\colon \mathbb{R}^n \to \mathbb{R}^n, x \mapsto (\tilde{f}(x_1), \ldots, \tilde{f}(x_n))$. Die auszugebende Funktion `f_out()` hat zwei Argumente, einen *double*-Vektor `x` der Länge ≥ 1 und einen einzelnen Wahrheitswert `nearest`. Falls `f_out()` mit `nearest=FALSE` aufgerufen wird, soll `f_in(x)` zurückgegeben und sowohl `x` als auch `f_in(x)` für spätere Verwendung gespeichert werden. Falls `f_out()` mit `nearest=TRUE` aufgerufen wird, soll für jedes Element x_i des übergebenen Vektors `x` aus den gespeicherten Werten das nächste x_j (kleinster Abstand $|x_i - x_j|$) gefunden und ein Vektor aus den entsprechenden $f(x_j)$-Werten ausgegeben werden.

Nutze für diese Aufgabe keine Schleifen! Funktionen wie `apply()` und `outer()` können hilfreich sein.

```
# Example 1
f_out <- mem_near(sin)
f_out(c(0, pi, 2*pi), nearest=FALSE)
## [1]  0.000000e+00  1.224606e-16 -2.449213e-16
f_out(c(pi/2, 3*pi/2), nearest=FALSE)
## [1]  1 -1
f_out(0:4, nearest=TRUE)
```

```
## [1]  0.000000e+00  1.000000e+00  1.000000e+00  1.224606e-16 -1.000000e+00
f_out(0:4, nearest=FALSE)
## [1]  0.0000000  0.8414710  0.9092974  0.1411200 -0.7568025
f_out(0:4, nearest=TRUE)
## [1]  0.0000000  0.8414710  0.9092974  0.1411200 -0.7568025
```

```
# Example 2
f_in <- function(x) x^2
f_out <- mem_near(f_in)
xl <- seq(-2, 2, len=300)
plot(xl, f_in(xl), type='l')
grid()
x <- -2:2
points(x, f_out(x, FALSE), col=2, pch=3, lwd=3, cex=2)
lines(xl, f_out(xl, TRUE), col=3, lwd=2)
set.seed(0)
x <- runif(20, min=-2, max=2)
points(x, f_out(x, FALSE), col=4, pch=3, lwd=3, cex=2)
lines(xl, f_out(xl, TRUE), col=5, lwd=2)
```

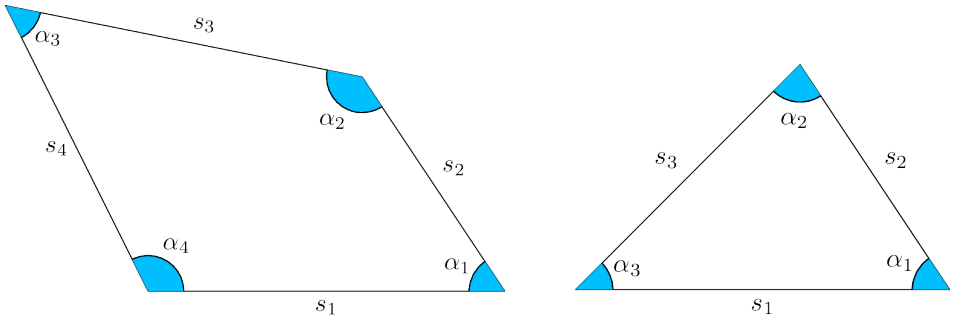


S3-Polygone (30 Punkte)

Ein Objekt der S3-Klasse `Polygon` sei eine Liste mit den Einträgen `sides` und `angles`, die jeweils numerische Vektoren der gleichen Länge sind. Dabei sind `sides` die (positiven) Längen der Seiten und `angles` die Winkel (zwischen 0 und 2π). Dabei ist `angles[i]` der Winkel zwischen `sides[i]` und `sides[i+1]` bzw zwischen

`sides[length(sides)]` und `sides[1]`.

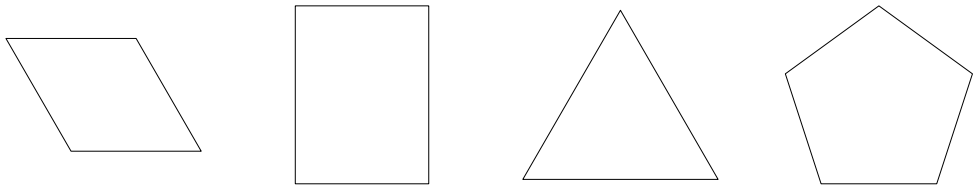
Wir wollen außerdem die Unterklassen `Quadrilateral` (Viereck) und `Triangle` (Dreieck) betrachten. `Quadrilateral` hat außerdem die Unterklasse `Rectanlge` (Rechteck).



a) Die Funktion `plot_polygon()` zeichnet ein gegebenes Polygon. Sorge dafür, dass dies auch direkt beim Aufruf von `plot()` mit einem Polygon-Objekt passiert.

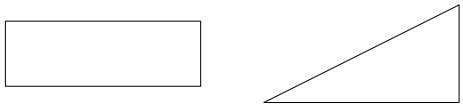
```r
# a) ---------------------------------------------------------------------------

poly1 <- structure(
  list(sides = c(2, 2, 2, 2), angles = c(pi/3, 2*pi/3, pi/3, 2*pi/3)),
  class = c("Quadrilateral", "Polygon"))
poly2 <- structure(
  list(sides = c(3, 4, 3, 4), angles = rep(pi/2, 4)),
  class = c("Rectangle", "Quadrilateral", "Polygon"))
poly3 <- structure(
  list(sides = c(2, 2, 2), angles = rep(pi/3, 3)),
  class = c("Triangle", "Polygon"))
poly4 <- structure(
  list(sides = rep(1, 5), angles = rep(3/5*pi, 5)),
  class = c("Polygon"))

get_corners <- function(x) {
  n <- length(x$sides)
  if (n < 3) return(invisible(NULL))
  out <- matrix(ncol = 2, nrow = length(x$sides))
  direction <- c(1, 0)
  point <- c(0, 0)
  out[1, ] <- point
  for (i in 1:(n-1)) {
    point <- point + direction * x$sides[i]
    out[i+1, ] <- point
    a <- pi - x$angles[i]
    direction <- matrix(c(cos(a), sin(a), -sin(a), cos(a)), ncol=2) %*% direction
  }
  out
}

plot_polygon <- function(x) {
  corners <- get_corners(x)
  plot(range(corners[,1]), range(corners[,2]), type = 'n', axes=FALSE, ann=FALSE, asp=1)
  polygon(corners[,1], corners[,2], lwd = 3)
}
```

```
plot(poly1)
plot(poly2)
plot(poly3)
plot(poly4)
```



b) Schreibe eine Funktion `Rectangle(w, h)`, die ein `Rectangle`-Objekt mit Breite `w` und Höhe `h` erstellt. Schreibe eine Funktion `Triangle(s1, s2, a1)`, die ein `Triangle`-Objekt mit `s1` gleich `sides[1]`, `s2` gleich `sides[2]` und `a1` gleich `angles[1]` erstellt.

```
plot(Rectangle(3, 1))
plot(Triangle(2, 1, pi/2))
```



c) Schreibe eine Funktion `validate_Triangle(x)`, die überprüft, ob `x` ein `Triangle` ist, und insbesondere, ob die Winkel und Seiten zusammenpassen.

```
p <- Triangle(2, 1, pi/2)
validate_Triangle(p) # ok
p$angles[1:2] <- p$angles[2:1]
validate_Triangle(p)
## Error in validate_Triangle(p): sides and angles do not fit
```

d) Schreibe eine generische Funktionen `circumference(x)`, die für Polygone ihren Umfang ausgibt.

```
abs((sqrt(5)+2+1) - circumference(Triangle(2, 1, pi/2))) < 1e-14
## [1] TRUE
circumference(Rectangle(3, 1))
## [1] 8
```

e) Schreibe eine generische Funktionen `area(x)`, die für Dreiecke und Vierecke ihre Fläche ausgibt.

```
area(Triangle(2, 1, pi/2))
## [1] 1
area(Rectangle(3, 1))
## [1] 3
area(poly1)
## [1] 3.464102
area(poly2)
## [1] 12
area(poly3)
## [1] 1.732051
```

Hinweis: https://de.wikipedia.org/wiki/Dreieck#Formeln und https://de.wikipedia.org/wiki/Viereck#Formeln

R6-GridPath (20 Punkte)

Erstelle eine R6-Klasse `GridPath`, die es erlaubt einen Pfad auf einem karierten Feld zu zeichnen, indem wir Objekten der Klasse Bewegungskommandos geben.

a) `GridPath` hat zwei Felder `path` – eine $(n \times 2)$-Matrix – und `dir` – ein *character*-Vektor der Länge 1. In `path` wird ein Pfad bestehend aus Koordinaten in $(x, y) \in \mathbb{N}^2$ gespeichert, in `dir` die aktuelle Blickrichtung als Wert `"N"`, `"E"`, `"S"`, oder `"W"` für die 4 Himmelsrichtungen. Die Klasse hat die drei Methoden `move()`, `rotate_left()` und `rotate_right()`. Die `rotate_`-Funktionen ändern den aktuellen Wert von `dir` in einen neuen um. Die Methoden sollen einen Rückgabewert haben, der das sogenannte *Method-Chaining* erlaubt, siehe Beispiel.

```
gpath1 <- GridPath$new()
gpath1$dir <- "N"
gpath1$rotate_left()
gpath1$dir
## [1] "W"
gpath1$rotate_left()
gpath1$dir
## [1] "S"
gpath1$rotate_right()$rotate_right()$rotate_right() # method chaining
gpath1$dir
## [1] "E"
```

Die Methode `move()` hat ein Argument `steps` – eine positive ganze Zahl. Damit werden `path` entsprechend viele Einträge hinzugefügt, die die Koordinaten bei Bewegung in die in `dir` gespeicherte Richtung enthalten.

```
gpath1$path <- matrix(c(0, 0), nrow=1)
gpath1$dir <- "N"
gpath1$move(3)
gpath1$path
##     [,1] [,2]
##        0    0
## cur    0    1
## cur    0    2
## cur    0    3
gpath1$rotate_right()$move(2)$rotate_right()$move(1)
gpath1$path
##     [,1] [,2]
##        0    0
## cur    0    1
## cur    0    2
## cur    0    3
## cur    1    3
## cur    2    3
## cur    2    2
```

Folgende Liste gibt die Koordonatenveränderung bei Fortbewegung in die entsprechende Himmelsrichtung an.

```
list(
  N = c(0, 1),
  S = c(0, -1),
```

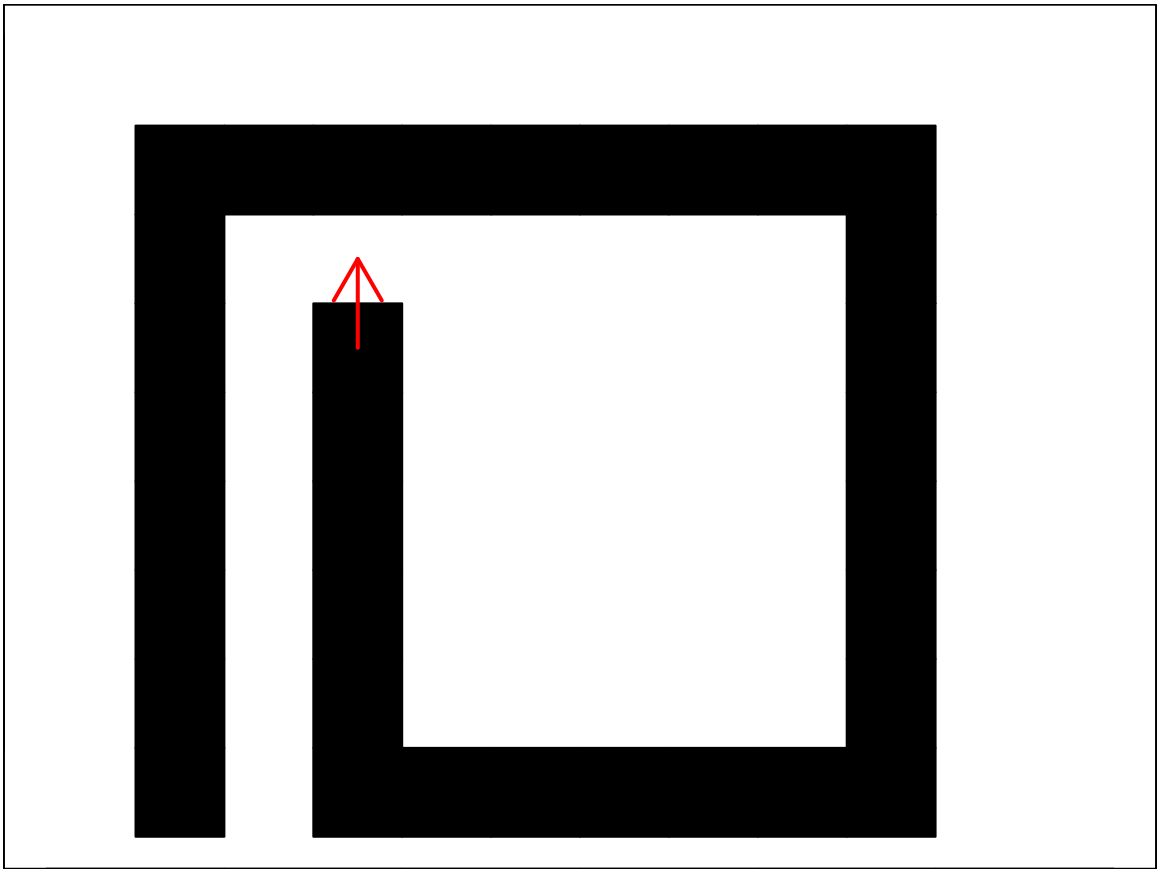
```
  W = c(-1, 0),
  E = c(1, 0)
)
```

b) Füge der Klasse eine `initialize()`-Methode mit einem Argument `dir` hinzu, die das Feld `dir` auf `dir` setzt und das Feld `path` mit `matrix(c(0, 0), nrow=1)` initialisiert. Hat eine R6-Klasse eine Methode mit dem Namen `initialize()`, so dient diese als Konstruktor und wird beim Erzeugen eines Objektes aufgerufen.

```
gpath2 <- GridPath$new("W") # calls initialize("W")
gpath2$dir
## [1] "W"
gpath2$path
##      [,1] [,2]
## [1,]    0    0
```

c) Füge der Klasse eine `print()`-Methode hinzu. Diese soll den Pfad und die aktuelle Richtung als Plot ausgeben. Dazu können zB die Funktionen `arrow()` und `plot(NA, xlim=<...>, ylim=<...>, asp=1)`, `rect()` oder `image(mat, useRaster=TRUE, asp=1)` hilfreich sein.

```
gpath3 <- GridPath$new('N')
gpath3$move(7)$
  rotate_right()$move(8)$
  rotate_right()$move(7)$
  rotate_right()$move(6)$
  rotate_right()$move(5)
gpath3 # calls print(gpath3) calls print.R6(gpath3) calls gpath3$print()
```



Bemerkung: Bei weiterem Interesse an R6-Klassen lohnt es sich insbesondere über private und aktive Felder

zu lernen, zB hier: https://adv-r.hadley.nz/r6.html.