Programowanie w języku R. Programowanie funkcyjne w R

Aleksander Denisiuk
Uniwersytet Warmińsko-Mazurski
Olsztyn, ul. Słoneczna 54
denisjuk@matman.uwm.edu.pl

30 maja 2020

Programowanie funkcyjne w R

Najnowsza wersja tego dokumentu dostępna jest pod adresem

`http://wmii.uwm.edu.pl/~denisjuk/uwm`

Programowanie funkcyjne

Wstęp

- Języki programowania funkcyjnego
  - czyste funkcje
- Trzy wzorce programistyczne
  - funkcjonał
  - fabryka funkcji
  - operatory funkcjonalne

Funkcjonał

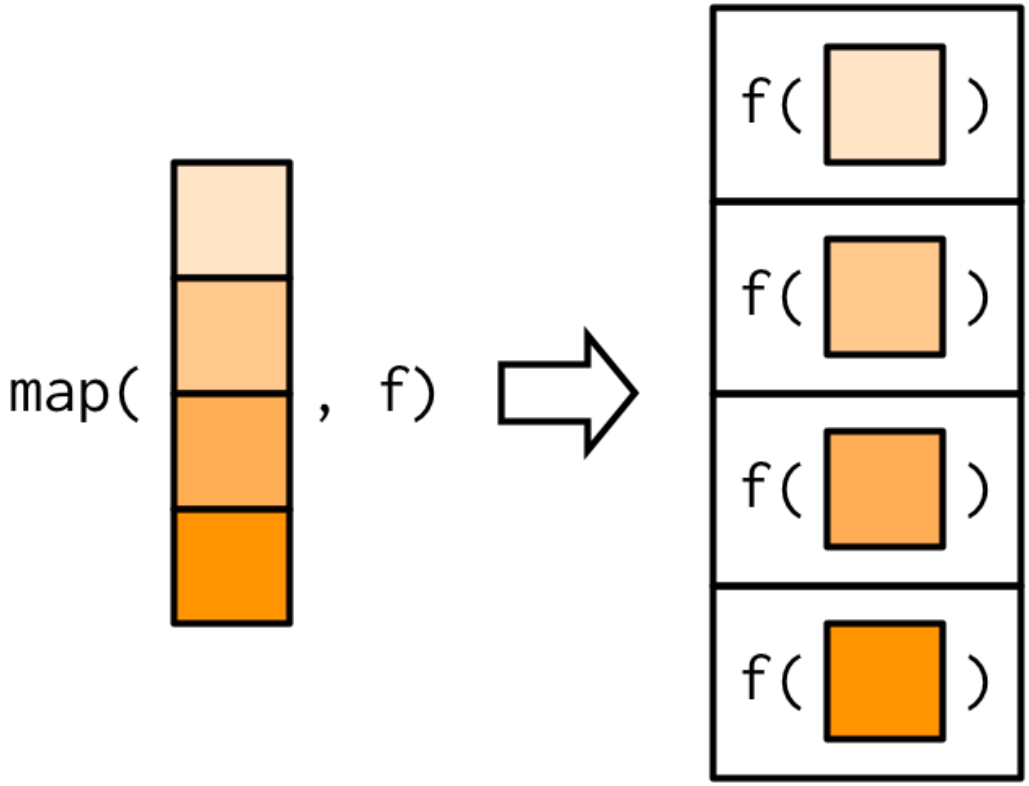
Funkcjonały

- Zastosować jedną funkcję do wielu danych wejściowych
- Zastępuje wielokrotne pętle `for` wywołaniem jednej funkcji, którja bierze jako argument funkcję
  - na przykład, `lapply`
- Najważniejszy wzorzec analizy danych (Hadley Wickham)
- Przykład:

```
randomise <- function(f) f(runif(1e3))

randomise(mean)
randomise(sum)
```

purrr::map()

- Na wejściu wektor i funkcja
- Na wyjściu lista



- Analogicznie do `lapply()`

Przykład


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

Wynik — wekor

- Na wyjściu wektor: `map_lgl()`, `map_int()`, `map_dbl()`, `map_chr()`

```
map_chr(mtcars, typeof)
map_lgl(mtcars, is.double)
n_unique <- function(x) length(unique(x))
map_int(mtcars, n_unique)
map_dbl(mtcars, mean)
```

map_*

- Na wejściu dowolny wektor
- Na wyjściu wektor tej samej długości
  - `f()` powinna produkować jedną wartość
    - odpowiedniego typu

W podstawowym R

- `sapply()`
  - zwraca listę, macierz albo wektor — jak się uda
- `vapply()`
  - pozwala na kontrolę zwracanego typu danych

```
map_dbl(x, mean, na.rm = TRUE)
vapply(x, mean, na.rm = TRUE,
                FUN.VALUE = double(1) )
```

Funkcje anonimowe


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

- Skrót w bibliotece `purrr`

```
map_dbl(mtcars, ~ length(unique(.x)))
```

Wybór kolumn listy

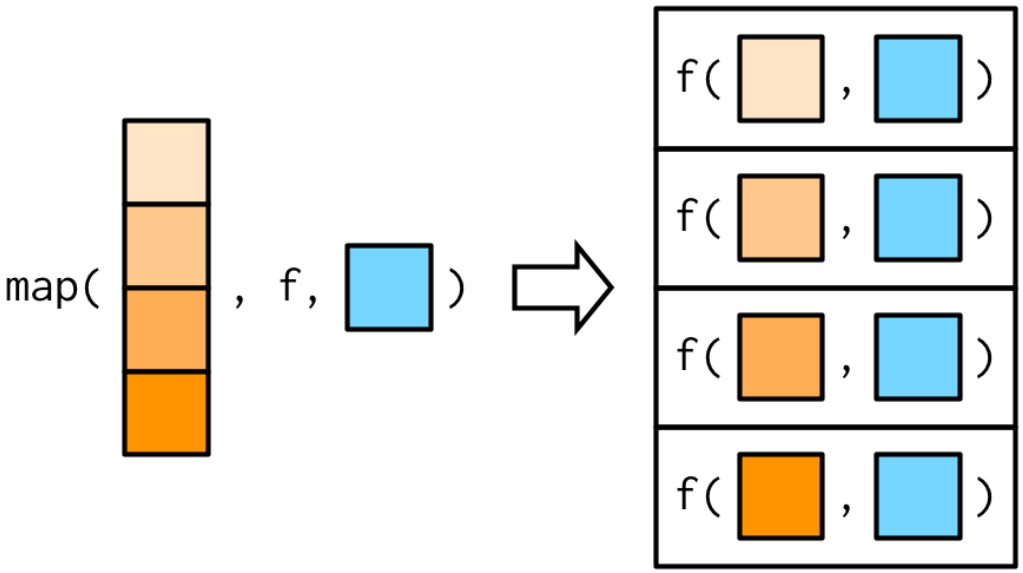
- Według nazwy bądź pozycji, albo nazwy i pozycji

```
x <- list(
  list(-1, x = 1, y = c(2), z = "a"),
  list(-2, x = 4, y = c(5, 6), z = "b"),
  list(-3, x = 8, y = c(9, 10, 11) )
)
map_dbl(x, "x")

map_dbl(x, 1)

map_dbl(x, list("y", 1))

map_chr(x, "z")
map_chr(x, "z", .default = NA)
```

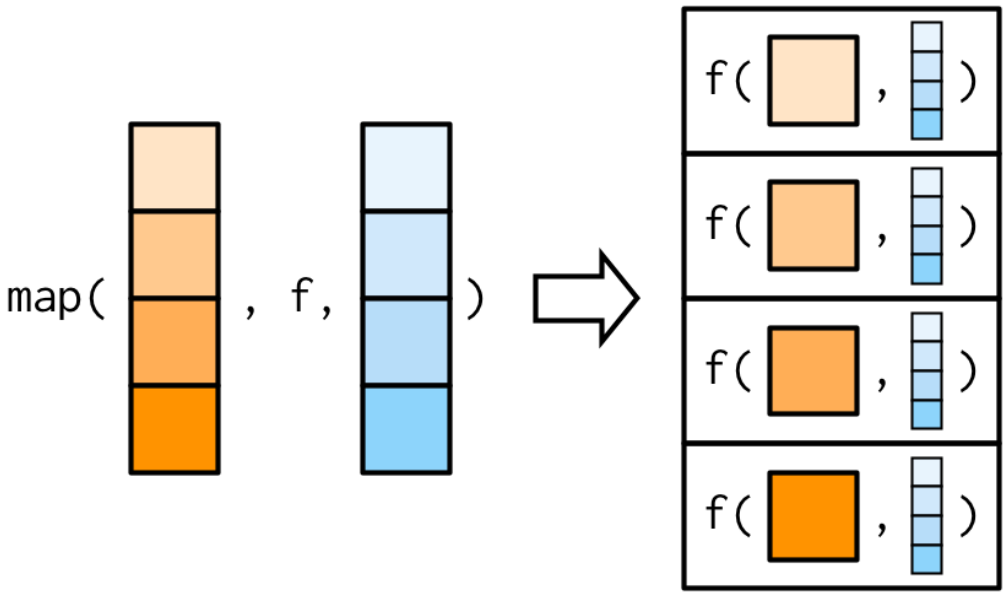
Przekazywanie argumentu przez . . .

- Dodatkowe argumenty funkcji

```
map_dbl(x, ~ mean(.x, na.rm = TRUE) )
map_dbl(x, mean, na.rm = TRUE)
```



Dodatkowy argument —wektor



Obliczenie dodatkowych argumentów

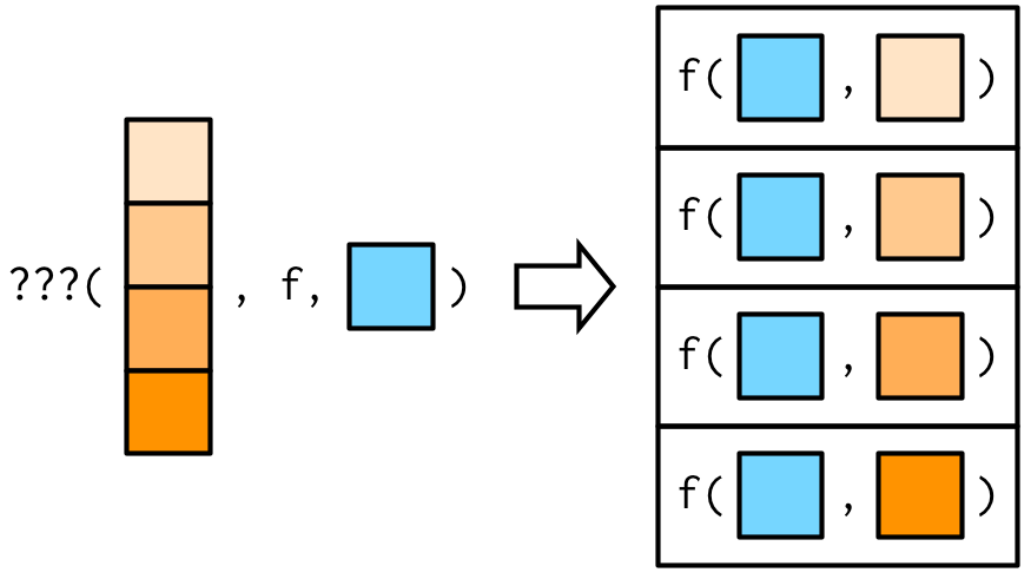
- Za każdym razem vs. jeden raz

```
plus <- function(x, y) x + y

x <- c(0, 0, 0, 0)

map_dbl(x, ~ plus(.x, runif(1)))
map_dbl(x, plus, runif(1))
```

Nazwy argumentów

- Podawać pełne nazwy: `map(x, mean, 0.1)` vs. `map(x, mean, trim = 0.1)`
- Argumenty funkcji `map()`
  - `.f`
  - `.x`

Wektor jako drugi argument



- Brak możliwości bezpośredniej
- Rozwiązania:
  - funkcja anonimowa, która zmienia kolejność argumentów
  - zasady dopasowywania argumentów

Przykład


```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(1000)

map_dbl(trims, ~ mean(x, trim = .x))
map_dbl(trims,
        function(trim) mean(x, trim = trim))

map_dbl(trims, mean, x = x)
```

Uogólnienia map()

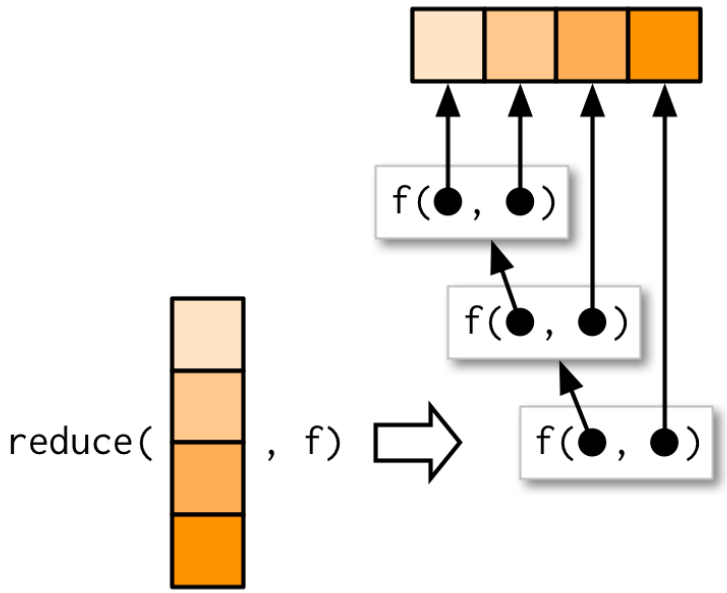
Argument \ Wynik	lista	wektor	ten sam typ	brak
jeden	`map()`	`map_lgl()`,...	`modify()`	`walk()`
dwa	`map2()`	`map2_lgl()`,...	`modify2()`	`walk2()`
jeden + indeks	`imap()`	`imap_lgl()`,...	`imodify()`	`iwalk()`
N	`pmap()`	`pmap_lgl()`,...	—	`pwalk()`

W podstawowym R

- `Map()`
  - wszystkie argumenty są wektorami tej samej długości
- `mapply()`
  - wielowymiarowa wersja `sapply()`

purrr::reduce()

- `reduce()` bierze wektor długości n i produkuje wektor długości 1, obliczając dwuargumentową funkcję
  - przykładowo `reduce(1:4, f)` $\Longleftrightarrow$ `f(f(f(1, 2), 3), 4)`



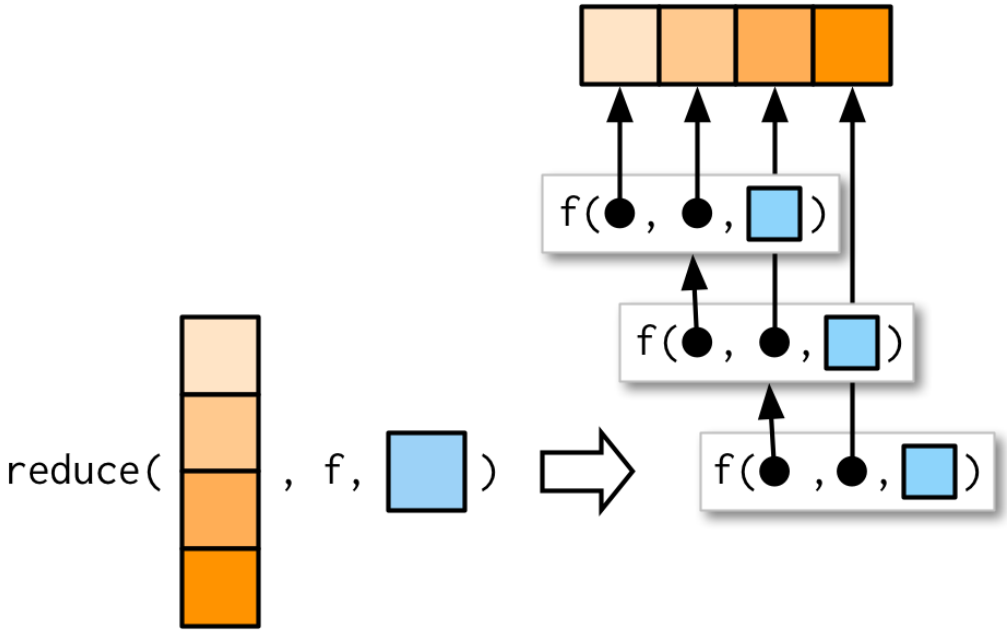
- uogólnić dwuargumentową funkcję na wiele agrumentów

Przykład

- Dana jest lista wektorów. Znaleźć wspólne elementy

```
l <- map(1:4, ~ sample(1:10, 15, replace = T))
str(l)

out <- l[[1]]
out <- intersect(out, l[[2]])
out <- intersect(out, l[[3]])
out <- intersect(out, l[[4]])
out

reduce(l, intersect)
```

Dodatkowe argumenty



W podstawowym R

- `Reduce()`
  - inna kolejność argumentów

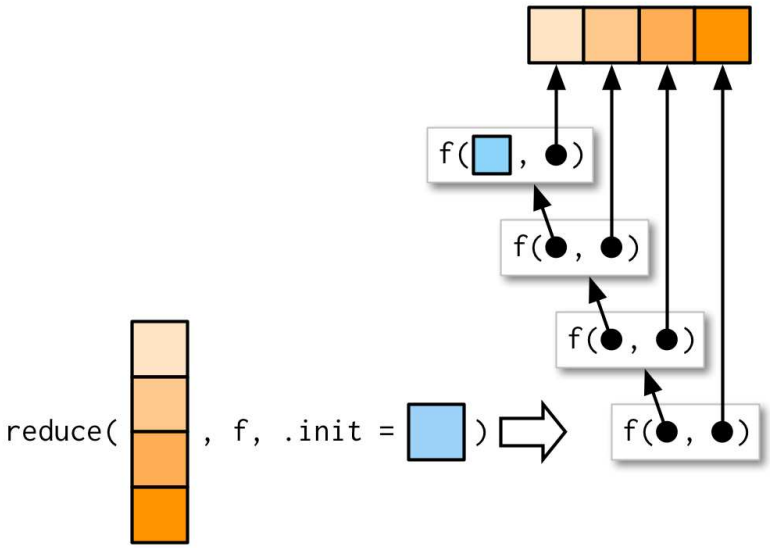
purrr::accumulate()

- `accumulate()` działa jak `reduce(`, ale wyświetla pośrednie wyniki

```
accumulate(l, intersect)
```

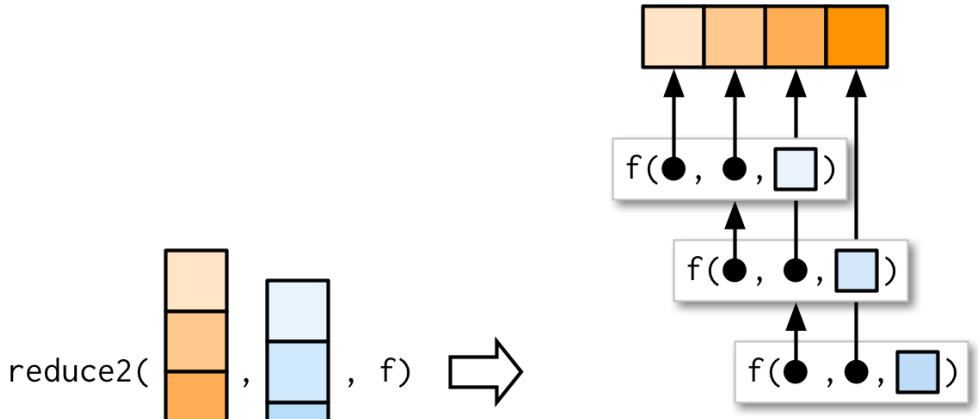
Wartość początkowa

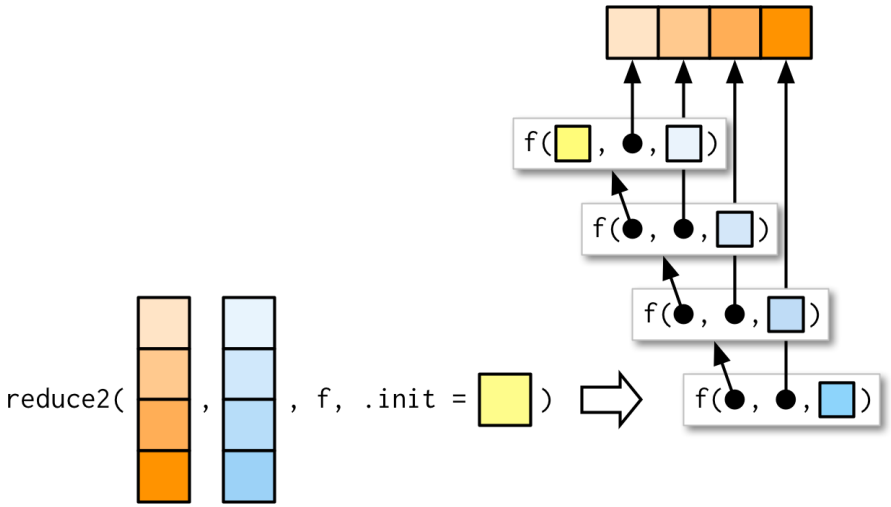

```
reduce(1, `+`)
reduce("a", `+`)
reduce(integer(), `+`)
```




```
reduce(integer(), `+`, .init = 0)
reduce("a", `+`, .init = 0)
```

purrr::reduce2()





Predykaty

- Zwracają wartości **`TRUE`** albo **`FALSE`**
- Predykaty-funkcjonały w purrr:
  - `some(.x, .p)` oraz `every(.x, .p)`
  - `detect(.x, .p)` — pierwsza wartość, `detect_index(.x, .p)` — pierwszy indeks
  - `keep(.x, .p)` — zostawia, `discard(.x, .p)` — usuwa wartości pasujące do `.p`
    - `.p` — funkcja-predykat

Przykład


```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))

detect(df, is.factor)

detect_index(df, is.factor)

str(keep(df, is.factor))

str(discard(df, is.factor))
```

Odmiany `map()`


```
df <- data.frame(
  num1 = c(0, 10, 20),
  num2 = c(5, 6, 7),
  chr1 = c("a", "b", "c"),
  stringsAsFactors = FALSE
)

str(map_if(df, is.numeric, mean))
str(modify_if(df, is.numeric, mean))
str(map(keep(df, is.numeric), mean))
```

Inne funkcjonały w podstawowym R

- `apply` działa na macierzach i wielowymiarowych tablicach
- Funkcjonały matematyczne:
  - `integrate()` — całkowanie
  - `uniroot()` — znalezienie pierwiastków
  - `optimise()` — znalezienie ekstremów

Fabryka funkcji

Wstęp

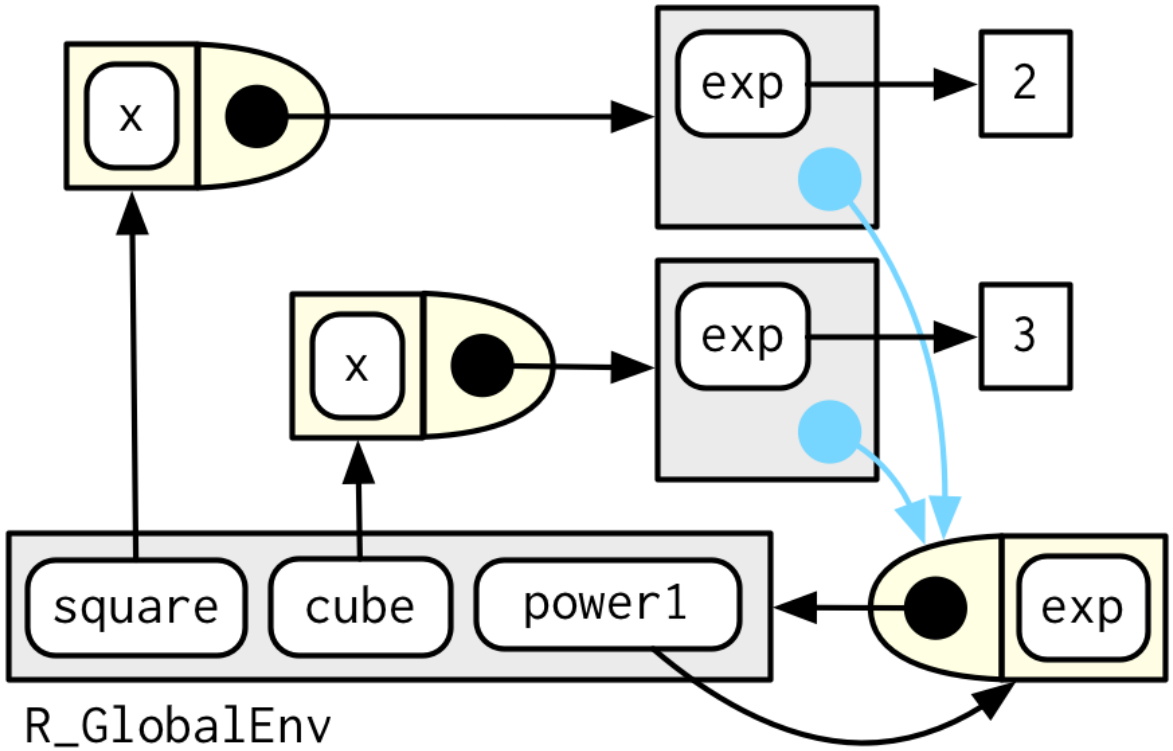
- Funkcja, która generuje funkcje (domknięcie, *closure*)
- Przykładowo:

```
power1 <- function(exp) {
  function(x) {
    x ^ exp
  }
}

square <- power1(2)
cube <- power1(3)

square(3)
cube(3)
```

Środowisko leksyczne

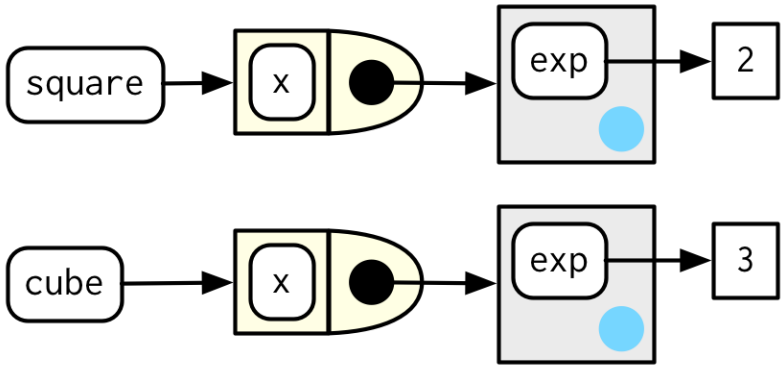

```
square
cube

library(rlang)
env_print(square)
env_print(cube)
```

Diagramy

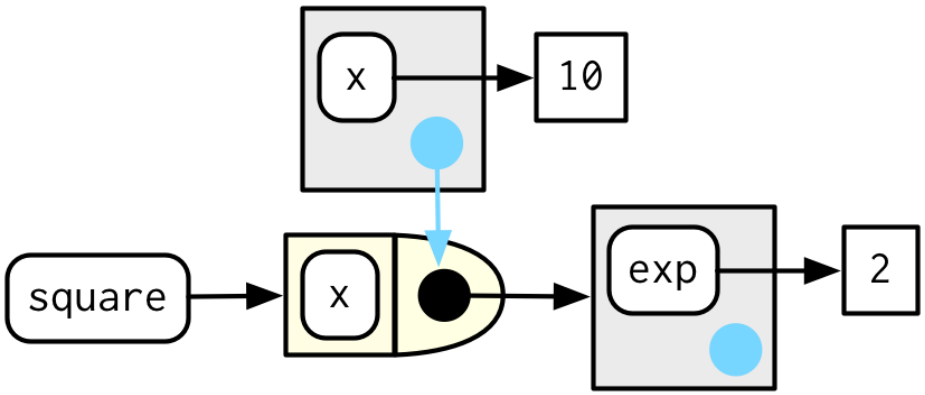


- Środowisko leksyczne *domknięcia* == środowisko wykonania *fabryki*

Przykład




```
square(10)
```



Wartościowanie leniwe

- Co wyświetli kod:

```
x <- 2
square <- power1(x)
x <- 3

square(2)
# ??
```

Poprawka


```
power2 <- function(exp) {
  force(exp)
  function(x) {
    x ^ exp
  }
}

x <- 2
square <- power2(x)
x <- 3
square(2)
```

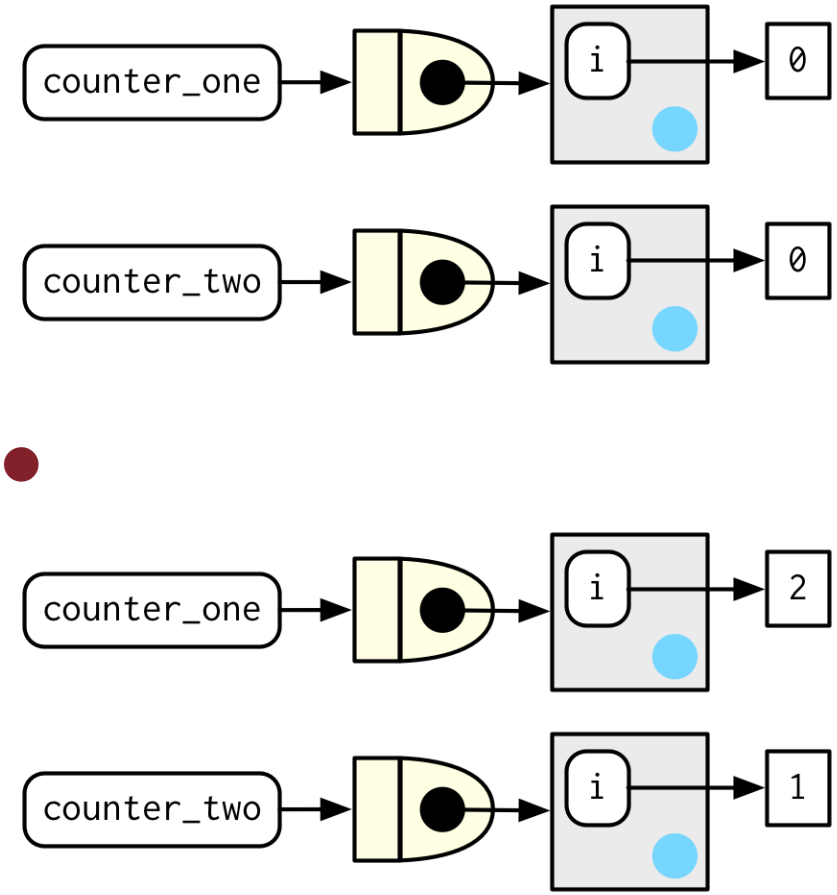
Funkcje statyczne

- Operator `<<-` — zmienia zmienną w środowisku nadrzędnym

Przykład


```
new_counter <- function() {
  i <- 0

  function() {
    i <<- i + 1
    i
  }
}

counter_one <- new_counter()
counter_two <- new_counter()

counter_one()
counter_one()
counter_two()
```

Diagramy



Odśmiecanie pamięci

- Zmienne z zewnętrznego środowiska leksykalnego domknięcia są *osiągalne*
  - ✦ nie będą usunięte przy odśmiecaniu pamięci

Biblioteka `scales`

- Fabryka funkcji

```
library(scales)
y <- c(12345, 123456, 1234567)

comma_format()(y)
number_format(scale = 1e-3, suffix = " K")(y)
```

Etykiety w ggplot


```
library(ggplot2)
df <- data.frame(x = 1, y = y)
core <- ggplot(df, aes(x, y)) +
  geom_point() +
  scale_x_continuous(breaks = 1, labels = NULL) +
  labs(x = NULL, y = NULL)
core
core + scale_y_continuous(
  labels = comma_format()
)
core + scale_y_continuous(
  labels = number_format(scale = 1e-3,
  suffix = " K")
)
core + scale_y_continuous(
  labels = scientific_format()
)
```

Zróżnicowane komórki w histogramach


```
library(ggplot2)
sd <- c(1, 5, 15)
n <- 100

df <- data.frame(x = rnorm(3 * n, sd = sd),
    sd = rep(sd, n))

ggplot(df, aes(x)) +
  geom_histogram(binwidth = 2) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```

- Każdy słupek histogramu ma taką samą ilość obserwacji
- Szerokość jest bardzo zróżnicowana

Zróżnicowane komórki w histogramach

- Funkcja `binwidth_bins()` oblicza ilość obserwacji

```
binwidth_bins <- function(n) {
  force(n)

  function(x) {
    (max(x) - min(x)) / n
  }
}

ggplot(df, aes(x)) +
  geom_histogram(binwidth = binwidth_bins(20)) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```

Optymalna szerokość


```
base_bins <- function(type) {
  fun <- switch(type,
    Sturges = nclass.Sturges,
    scott = nclass.scott,
    FD = nclass.FD,
    stop("Unknown type", call. = FALSE)
  )
  function(x) {
    (max(x) - min(x)) / fun(x)
  }
}
ggplot(df, aes(x)) +
  geom_histogram(binwidth = base_bins("FD")) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```

Przekształcenie Boksa-Coksa (Boxa-Coxa)

- Aka Power Transform

$$y^\lambda = \begin{cases} \dfrac{x^\lambda - 1}{\lambda} & \text{jeżeli } \lambda \neq 0, \\ \ln x & \text{jeżeli } \lambda = 0, \end{cases}$$

```
boxcox1 <- function(x, lambda) {
  stopifnot(length(lambda) == 1)

  if (lambda == 0) {
    log(x)
  } else {
    (x ^ lambda - 1) / lambda
  }
}
```

Fabryka funkcji


```
boxcox2 <- function(lambda) {
  if (lambda == 0) {
    function(x) log(x)
  } else {
    function(x) (x ^ lambda - 1) / lambda
  }
}
```

Wykresy


```
stat_boxcox <- function(lambda) {
  stat_function(aes(colour = lambda),
   fun = boxcox2(lambda), size = 1)
}

ggplot(data.frame(x = c(0, 5)), aes(x)) +
  lapply(c(0.5, 1, 1.5), stat_boxcox) +
  scale_colour_viridis_c(limits = c(0, 1.5))

ggplot(data.frame(x = c(0.01, 1)), aes(x)) +
  lapply(c(0.5, 0.25, 0.1, 0), stat_boxcox) +
  scale_colour_viridis_c(limits = c(0, 1.5))
```

Metoda Bootstrap


```
boot_permute <- function(df, var) {
  n <- nrow(df)
  force(var)

  function() {
    col <- df[[var]]
    col[sample(n, replace = TRUE)]
  }
}

boot_mtcars1 <- boot_permute(mtcars, "mpg")
head(boot_mtcars1())
head(boot_mtcars1())
```

Operator funkcjonalny

Wstęp

- Funkcja bierze jako argument funkcje, zwraca funkcje
  - *Dekoracja* funkcji

```
chatty <- function(f) {
  force(f)

  function(x, ...) {
    res <- f(x, ...)
    cat("Processing ", x, "\n", sep = "")
    res
  }
}
f <- function(x) x ^ 2
s <- c(3, 2, 1)

purrr::map_dbl(s, chatty(f))
```

Problem

- Wersja iteracyjna zwraca wynik

```
x <- list(
  c(0.512, 0.165, 0.717),
  c(0.064, 0.781, 0.427),
  c(0.890, 0.785, 0.495),
  "oops"
)

out <- rep(NA_real_, length(x))
for (i in seq_along(x)) {
  out[[i]] <- sum(x[[i]])
}
out
```

- Funkcjonał — nie:

```
map_dbl(x, sum)
```

Przechwytywanie błedów z purrr::safely()

- `purrr::safely()` zwraca listę z dwóch elemetów `result` i `error`
  - jeżeli obliczenia skończyły się pomyśnie, `result` zawiera wynik, `error == NULL`
  - jeżeli obliczenia skończyły się z błędem, `result == NULL`, `error` zawiera błąd

```
out <- map(x, safely(sum))
str(out)

out <- transpose(map(x, safely(sum)))
str(out)
```

Analiza wyników

- Znaleźć prawidłowe wyniki
- Zobaczyć dane, na których był błąd
- Filtrować wyniki

```
ok <- map_lgl(out$error, is.null)
ok
```

```
x[!ok]
```

```
out$result[ok]
```

Przykład

- Regresja liniowa z `glm()` może się skończyć niepowodzeniem

```
fit_model <- function(df) {
  glm(y ~ x1 + x2 * x3, data = df)
}

models <- transpose(map(datasets,
   safely(fit_model)))
ok <- map_lgl(models$error, is.null)

datasets[!ok]

models[ok]
```

Inne operatory z biblioteki purrr

- `possibly()` w przypadku błędu zwraca domyślną wartość
- `quietly()` przekształca komunikaty o błędzie i ostrzeżenia w dane wyjściowe
- `auto_browser()` automatycznie odpala `browser()` w kontekście funkcji, gdzie zaistniał błąd

Cache'owanie z `memoise::memoise()`

- Wolna funkcja

```
slow_function <- function(x) {
  Sys.sleep(1)
  x * 10 * runif(1)
}
system.time(print(slow_function(1)))
system.time(print(slow_function(1)))
```

- Wersja z cache'owaniem

```
library(memoise)
fast_function <-
      memoise::memoise(slow_function)

system.time(print(fast_function(1)))
system.time(print(fast_function(1)))
```

Przykład: liczby Fibonacciego

- Wersja rekurencyjna

```
fib <- function(n) {
  if (n < 2) return(1)
  fib(n - 2) + fib(n - 1)
}
system.time(fib(23))
system.time(fib(24))
```

- Wersja z cache'owaniem

```
fib2 <- memoise::memoise(function(n) {
  if (n < 2) return(1)
  fib2(n - 2) + fib2(n - 1)
})
system.time(fib2(23))
system.time(fib2(24))
```

Uwaga

- Wersja z cache'owaniem ma efekty uboczne
  - nie jest funkcją *czystą*

Pobieranie danych z serwera

- Dana jest lista adresów URL
  - ✦ pobrać wszystkie na dysk
- Proste rozwiązanie:

```
urls <- c(
  "automobiles" = "http://archive.ics.uci.edu/ml/
  "adults" = "https://archive.ics.uci.edu/ml/mach
## i tak dalej
)
path <- paste0(names(urls), ".data")

walk2(urls, path, download.file, quiet = TRUE)
```

Pobieranie danych z serwera: dekoracja

- Dodać krótką pauzę między pobieraniami
- Wyświetlić kropkę po co dziesiątym pobieraniu
- Rozwiązanie z pętlą:

```
for(i in seq_along(urls)) {
  Sys.sleep(0.1)
  if (i %% 10 == 0) cat(".")
  download.file(urls[[i]], path[[i]])
}
```

  - ✦ Podzielić dekoratory, zaimplementować jako operatory funkcjonalne

Dodanie pauzy


```
delay_by <- function(f, amount) {
  force(f)
  force(amount)

  function(...) {
    Sys.sleep(amount)
    f(...)
  }
}
```

- Sprawdzamy:

```
system.time(runif(100))
system.time(delay_by(runif, 0.1)(100))
walk2(urls, path, delay_by(download.file, 0.1),
      quiet = TRUE)
```

Wyświetlenie kropki

- Wykorzystamy domknięcie:

```
dot_every <- function(f, n) {
  force(f)
  force(n)
  i <- 0
  function(...) {
    i <<- i + 1
    if (i %% n == 0) cat(".")
    f(...)
  }
}
```

- Sprawdzamy:

```
walk(1:100, runif)
walk(1:100, dot_every(runif, 10))
```

Wyświetlenie kropki

- Zmiana pętli:

```
walk2(
 urls, path,
 dot_every(delay_by(download.file, 0.1), 10),
 quiet = TRUE
)
```

- Z wykorzystaniem potoków:

```
walk2(
  urls, path,
  download.file
     %>% dot_every(10)
     %>% delay_by(0.1),
  quiet = TRUE
)
```