tidysq

tidysq forum

o kopiach obiektów w R

Obiekty i środowiska

tidysq

- Typową praktyką programistyczną jest przypisywanie obiektu do zmiennej, aby można było później ponownie się do niego odwołać
- W języku R **zmienne** same w sobie **są obiektami** o typie `symbol`
  - `x ← "some variable" # naturalne występowanie obiektu typu symbol`
  - `quote(x) # ręczne uzyskiwanie obiektu typu symbol`

- Innym typem obiektu jest `environment` (środowisko)
- Służą do przechowywania obiektów w postaci **par symbol-wartość**
  - `x ← 2 # "x" jest symbolem, "2" jest wartością`

Dokładniej o środowiskach

tidysq

- Środowisko składa się z dwóch elementów: **listy par symbol-wartość** oraz **wskaźnika na *enclosing environment*** (środowisko-rodzica)
- W rezultacie środowiska tworzą **hierarchiczną**, drzewiastą strukturę

- **R przeszukuje środowiska** w poszukiwaniu wartości odpowiadającej symbolowi
- Jeżeli w danym środowisku nie znajdzie szukanego symbolu, **R przeszuka *enclosing environment***, powtarzając procedurę do uzyskania sukcesu
- Korzeniem hierarchii środowisk jest specjalny obiekt dostępny poprzez `emptyenv()`

Podstawowy mechanizm

tidysq

- Przypisanie obiektu do nowego symbolu tworzy **kopię obiektu w bieżącym środowisku**
  - ```
    x ← 1:6
    y ← x # przypisanie wartości do zmiennej y tworzy kopię x
    ```
- Przekazanie obiektu do funkcji tworzy **kopię obiektu w nowym środowisku**, utworzonym przez funkcję (tzw. *evaluation environment*)
  - ```
    x ← 1:6
    compute(x) # na potrzeby funkcji tworzona jest kopia zmiennej x
    ```

Jak jest naprawdę?

tidysq

Specyficzne zachowanie środowisk

tidysq

- W przeciwieństwie do “zwykłych” obiektów, **środowiska nie są kopiowane**
- Przypisanie środowiska do symbolu tak naprawdę **przypisuje jedynie referencję**
  - ```
    e1 ← .GlobalEnv # przypisanie referencji
    e2 ← e1 # skopiowanie referencji
    e2$x ← 2 # przypisanie wartości do zmiennej x w środowisku .GlobalEnv
    e1$x
    # 2
    ```
    ```

Leniwa ewaluacja

- Przypisanie zmiennej do nowego symbolu **nie kopiuje wartości** od razu
- Kopiowanie zachodzi dopiero w momencie, gdy któraś ze zmiennych **podlega modyfikacji**
  - ```
    x ← 1:6
    y ← x # kopiowanie nie zachodzi
    lobstr::ref(x) == lobstr::ref(y)
    # TRUE
    z ← x + 2 # x i y nadal wskazują na tę samą wartość
    y[2] ← 0 # w tym momencie następuje kopiowanie
    lobstr::ref(x) == lobstr::ref(y)
    # FALSE
    ```
- Zmieniana jest referencja zmiennej podlegającej modyfikacji
    ```

Obietnice z pokryciem

tidysq

- Kopiowanie za każdym razem byłoby **bardzo nieefektywne**
- W tym celu powstały obiekty typu `promise` (obietnice)
- W momencie wywołania funkcji **symbole** odpowiadające argumentom **są przypisane do obietnic**

- Obietnice składają się z trzech elementów:
  1. **wskaźnika na środowisko**, w którym ewaluowana jest funkcja
  2. **wyrażenia** opisującego wartość
  3. **wartości** (początkowo pustej)
- W momencie próby dostępu do wartości obietnicy **wyrażenie jest ewaluowane**, rezultat przypisywany do wartości i zwracany

A jak to działa w Rcpp?

Obiektowość Rcpp

- Rcpp służy do integracji C++ w R
- Obiekty Rcpp w C++ podlegają standardowym zasadom C++, a więc standardowo **kopiowana jest jedynie referencja**
- Istnieje funkcja `clone()` przeznaczona do **kopiowania wartości**
  - ```
    Rcpp::NumericVector x = {8, 13, 21};
    // poniższy kod kopiuje jedynie referencję (shallow copy)
    Rcpp::NumericVector y = x;
    // poniższy kod kopiuje wartości (deep copy)
    Rcpp::NumericVector z = clone(x);
    ```
    ```

Rcpp ma referencje

tidysq

- Wspomniana obiektowość Rcpp **działa również na obiekty przekazane z R**
- Obiekty przekazane do Rcpp zawierają referencje na oryginalne obiekty w R, dlatego **modyfikacje w Rcpp modyfikują też oryginalne obiekty**
  - ```
    x ← c("Hercules", "Poirot")
    # x może ulec zmianie w wyniku działania tej funkcji
    some_Rcpp_function(x)
    ```
    ```

Pułapka na nieświadomych

- Rcpp używa referencji do oryginalnych obiektów **tylko wtedy, kiedy nie następuje rzutowanie**
- Najczęściej rzutowanie zachodzi, gdy wektor w R jest typu `integer`
- Wówczas Rcpp rzutuje go do typu `Rcpp::NumericVector`
- Rzutowanie nie jest potrzebne, gdy wektor w R jest typu `numeric`

Pułapka na nieświadomych w praktyce

- Niech istnieje funkcja `increment_Rcpp( )`, która wszystkie wartości przekazanego wektora podnosi o 1
  - ```
    x_integer ← 1:5 # obiekt typu integer
    increment_Rcpp(x_integer)
    x_integer
    # [1] 1 2 3 4 5
    ```
  - ```
    x_numeric ← as.numeric(1:5) # obiekt typu numeric
    increment_Rcpp(x_numeric)
    x_numeric
    # [1] 2 3 4 5 6
    ```

tidysq

tidysq

by Mateusz Bąkała