Cheatography

r part6.1 Cheat Sheet

by Niki (worlddoit) via cheatography.com/170195/cs/36504/

Introduction

R is not a fully functional programming language because its functions are not pure, however it allows for a lot of functional programming by means of higher-order-functions

Higher-order-functions:

☑ Functions: take vectors as input and return vectors as output

☑ Functionals: take functions (and vectors) as input and return vectors as output. They allow to generalize and reapply techniques in data analysis to any number of inputs

☑ Function Factories: take vectors as input and return functions as output. These are used to create functions

☑ Function Operators: take functions as input and return functions as output. They are used to modify the behavior of functions

(**Basics)

Functionals

A functional is a function that takes functions as input and returns vectors as output.

`integrate(cos, 0, pi)`

`4.922505e-17 with absolute error < 2.2e-14`

They are a common alternative to for loops.

The basic syntax is `map(.x, .f, ...)` where

`.x` can be a list or any atomic vector

`.f` is a function that will be applied to each element of `.x`.

The code of `map` is very simple:

```
simple_map <- function(x, f, ...) {
out <- vector("list", length(x))
for (i in seq_along(x)) {
```

Functionals (cont)

```
out[[i]] <- f(x[[i]], ...) }
out}
```

In `purrr` there is a special syntax for anonymous functions

```
> purrr::map_dbl(mtcars, function(x) length(unique(x)))
```

is similar to

```
> purrr::map_dbl(mtcars, ~ length(unique(.x)))
```

The `map` extended family and friends is quite large:

details.

If all you want is to substitute the `for` loop, than the `base`

functionals of the `apply` family might be a better choice.

```
> x <- lapply(1:100, sqrt)
> apply(mtcars, 2, mean) #margin = 1 by row, 2 by col
```

If you can substitute the for loop with vectorization no need to use functionals.

(**Basics)

Functionals - Time Example

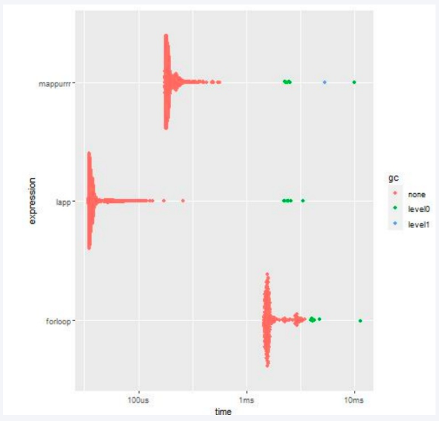
```
> (benchmap<-bench::mark(
+ forloop = { x <- vector("list",100)
+ for(i in seq_along(100))
x[[i]]<-sqrt(i)},
+ lapp = {x <- lapply (1:100, sqrt)},
+ mappurrr = { x <- purrr::map(1:100,sqrt)},
+ check = F
+ ))
# A tibble: 3 x 13
expression min median itr/sec
```

Functionals - Time Example (cont)

```
<bch:expr> <bch:tm> <bch:tm> <dbl>
1 forloop 1.44ms 1.57ms 580.
2 lapp 34.64us 36.47us 25570.
3 mappurrr 176.5us 188.17us 5054.
```

(***Advanced)

Functionals - Time Example photo



(***Advanced)

The apply family

ϟ `apply(X, MARGIN, FUN, ...)` returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

`apply(mat, 1, sum)`

ϟ `lapply` returns a list, each element of which is the result of applying a function to the corresponding element.

`lapply(x, mean)`

ϟ `rapply()` is a recursive version of `lapply()` with flexibility in how the result is structured.

```
> rapply(x, sqrt, how = "list")
> rapply(x, sqrt, how = "unlist")
```

r part6.1 Cheat Sheet

by Niki (worlddoit) via cheatography.com/170195/cs/36504/

The apply family (cont)

ϟ `sapply()` is a user-friendly version and wrapper of `lapply()` by
default returning a vector.

```
> sapply(1:5,sqrt)
```

ϟ `vapply` is similar to `sapply`, but has a pre-specified type of
return value, so it can be safer (and sometimes faster) to use.

```
> vapply(1:5, sqrt, numeric(1))
```

ϟ `mapply()` is a multivariate version of `sapply()` that applies a
function to all first (then second, third, and so on) elements of
its arguments. Arguments are recycled if necessary.

```
> mapply(rep, 1:2, 2:1)
```

ϟ `tapply()` applies a function to each (non-empty) group of
values given by a unique combination of the levels of certain
factors.

```
> with(dat, tapply(age, gender, mean))
```

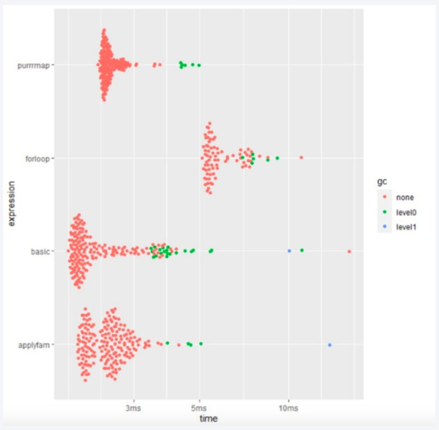
This could have also written as:

```
> dat %$% tapply(age, gender, mean)
```

`%$%` exposes the contents of the left-hand side object to the expression on the right.

(**Basics)

Time Performance of the different approaches



Friends of map

ϟ The `modify` friend of `map`

The `purrr::modify()` function tackles transformations that preserve the type input in the output.

```
> modify(df, ~ .x * 2)
```

Things are not modified in place but new objects are created

ϟ The `walk` friend of `map`

The `purrr::walk()` function tackles transformations that do not require to store the output but are focused only on side-effects

ϟ The `imap` friend of `map`

The `purrr::imap()` function and friends essentially mimic ways of looping:

```
> imap(x, ~ paste0("Label: ", .y, " Value: ", .x))
> imap_chr(x, ~ paste0("Label: ", .y, " Value: ", .x))
```

Predicate functionals

Predicate functions return `TRUE` or `FALSE`.
For instance, the testing functions is `.x()`.
Predicate functionals apply a predicate to the elements of a vector

```
> x<-list(1:2, c("a","b"), c(TRUE, FALSE))
> some(x, is.logical)
[1] TRUE
> every(x, is.vector)
[1] TRUE
> detect(x, is.logical); detect_index(x, is.logical)
[1] TRUE FALSE
[1] 3
> keep(x, is.character)
[[1]]
[1] "a" "b"
> discard(x, is.integer)
[[1]] [[2]]
[1] "a" "b" [1] TRUE FALSE
```

(**Basics)

Function Factories

Functions that make functions.

```
> log10
function (x) .Primitive("log10")
> changelog <- function(b) {
+ function(x) {
+ log(x)/log(b) }}
log10 <- changelog(10)
> log10(10)
[1] 1
> log10
function(x) {
```

Function Factories (cont)

```
log(x)/log(b)}
<environment:
0x000000000e46a9e8> #notice the
environment
```

The enclosing environment of the `log10` is the execution environments of the `changelog` function when `log10` was defined by assignment.

There is however a ‘bug’ due to lazy evaluation so it is better to always force the function parameter.

```
> changelog <- function(b) {
+ force(b)
+ function(x) {log(x)/log(b)}
+ }
```

You can combine factories with functionals

```
> names <- list(log2 = 2,
+ log3 = 3,
+ log10 = 10)
> (logs <- purrr::map(names,
changelog))
$log2
function(x) {log(x)/log(b)}
<bytecode: 0x000000000dbc6b98>
<environment:
0x000000000e03dc20>
> logs$log2(2)
[1] 1
> logs$log10(10)
[1] 1
> logs$log3(3)
[1] 1
```

You can use factories to pass different arguments according to your needs to other functions.

```
> n <- 100; sd <- c(1, 5, 15)
> df <- data.frame(x =
rnorm(3*n, sd = sd), sd =
rep(sd, n))
> histograms<-ggplot(df, aes(x))
+
+ geom_histogram(binwidth = 2) +
+ facet_wrap(~ sd, scales =
"free_x") +
```

Function Factories (cont)

```
+ labs(x = NULL)
> jpeg("histograms.jpeg")
> plot(histograms)
> dev.off()
null device
```

The code above creates `binwidth` facets in which the is the same, this is not ideal to compare the different histograms.

Output 1

we can create a variable bin width

```
> binwidth_bins <- function(n) {
+ force(n)
+ function(x) {
+ (max(x) - min(x)) / n
+ }
+ }
```

and then we run a new groups of histograms

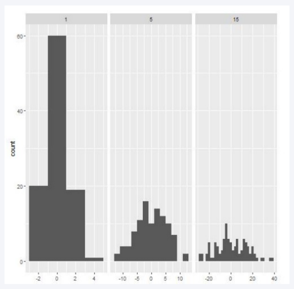
```
> histograms2<-ggplot(df,
aes(x)) +
+ geom_histogram(binwidth =
binwidth_bins(20)) +
+ facet_wrap(~ sd, scales =
"free_x") +
+ labs(x = NULL)
> jpeg("histograms2.jpeg")
> plot(histograms2)
> dev.off()
null device
```

Now the `binwidth` varies and keeps constant the number of observations in each bin

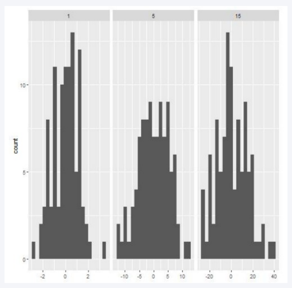
Output 2

(**Basics)

Output 1



Output 2



Function Operators

Functions that take functions as arguments and return other functions. They wrap a function and somehow extend its behavior without modifying the function output (decorator).

```
> log(10)
[1] 2.302585
> log("a")
Error in log("a") : non-numeric
argument to mathematical
function
> safe_log <- purrr::safely(log)
> safe_log
function (...)
capture_error(.f(...),
otherwise, quiet)
<bytecode: 0x000000001b790a88>
<environment:
0x000000000ed506f0>
> safe_log(10)
$result
[1] 2.302585
$error
```

r part6.1 Cheat Sheet

by Niki (worlddoit) via cheatography.com/170195/cs/36504/

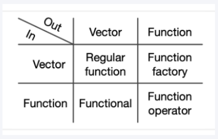
Function Operators (cont)

```
NULL
> safe_log("a")
$result
NULL
$error
<simpleError in
.Primitive("log")(x, base):
non-numeric argument to
mathematical function>
```

`safely()` is also useful in catching errors in the applications of functionals like `map`.

```
> out <- map(x, safely(sum))
> str(out)
List of 4
$ :List of 2
..$ result: int 10
..$ error : NULL
...
$ :List of 2
..$ result: NULL
..$ error :List of 2
.. ..$ message: chr "invalid
'type' (character) of argument"
.. ..$ call : language
.Primitive("sum")(..., na.rm =
na.rm)
.. ..- attr(*, "class")= chr
[1:3] "simpleError" "error"
"condition"
...
```

Summary



In \ Out	Vector	Function
Vector	Regular function	Function factory
Function	Functional	Function operator

resources: 1