Principles and Practice of Data Analysis

for Reproducible Research in R

The DRY Principle

Heather Turner

Department of Statistics, University of Warwick

2016–09–28

The DRY Principle

Analysis/reporting often involves performing very similar tasks, e.g. fitting different models to same data, fitting same model to different variables, etc.

A simple solution is to copy-paste code chunks and modify as necessary, however this has the following problems

- tedious to update, e.g. for new variable
- prone to error, e.g typos/not making all required modifications
- verbose code - harder to see what code is doing and what has changed

Good coding practice follows the DRY principle: Don't Repeat Yourself.

DRY Strategies

There are several techniques to avoid repeated code.

Some are *methods* are available for particular *classes* of objects, e.g.

- `update` to update the call in a model object, e.g. `"lm"`, and re-fit
- `%+%` to create a `"ggplot"` based on different data

Then there are techniques that can be applied to R chunks in general

- chunk reuse in rmarkdown/knitr documents
- custom functions to parameterize a chunk of R code
- apply/mapping functions to apply a function to different subsets of data
- iteration to repeat a chunk of code iterating over indices/elements of an object

Updating Models

Using `update` we can re-fit a model, only specifying the parts that need to change, e.g. to exclude an outlier, we could run

```
mod2 <- update(mod1, subset = id != 36)
```

To modify the formula, we can use `.` to refer to everything that was there before, e.g.

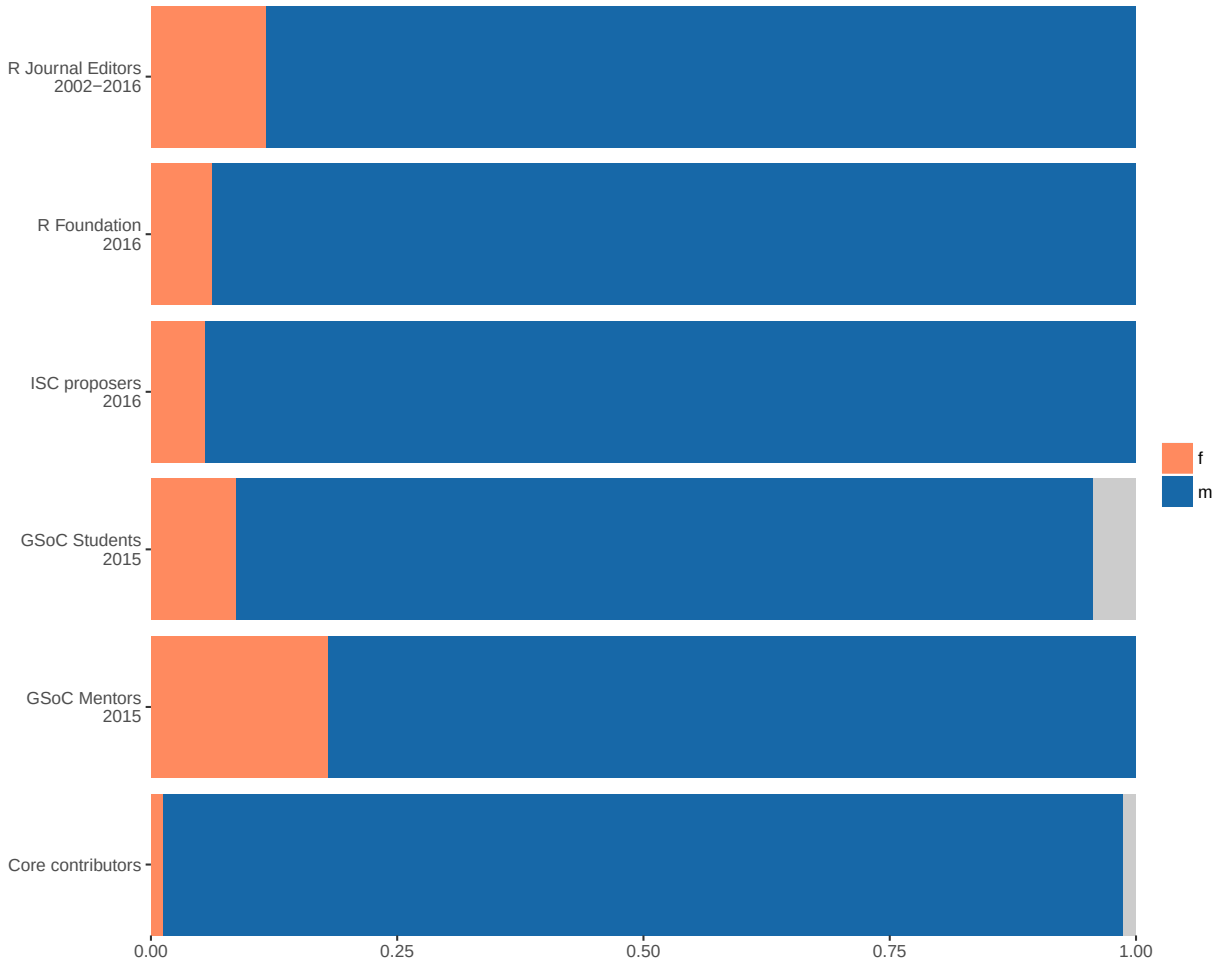
```
mod2 <- update(mod1, . ~ . - education + income)
mod2 <- update(mod1, scoreF ~ .)
```

Updating ggplots

```
library(rio)
library(ggplot2)
r_community <- import("Rcommunity.txt")
p <- ggplot(r_community, aes(x = Group, y = Percentage, fill = Gender)) +
    geom_bar(stat = "identity", position = "stack") + coord_flip() +
    scale_x_discrete(expand = c(0,0)) +
    scale_y_continuous(expand = c(0,0)) +
    scale_fill_manual(values = c("#FF8A5F", "#1768A8"),
                      na.value = "grey80") +
    xlab(NULL) +
    theme(panel.background = element_blank(),
          panel.grid.major = element_blank(),
          axis.title.x = element_blank(),
          legend.title = element_blank())
```

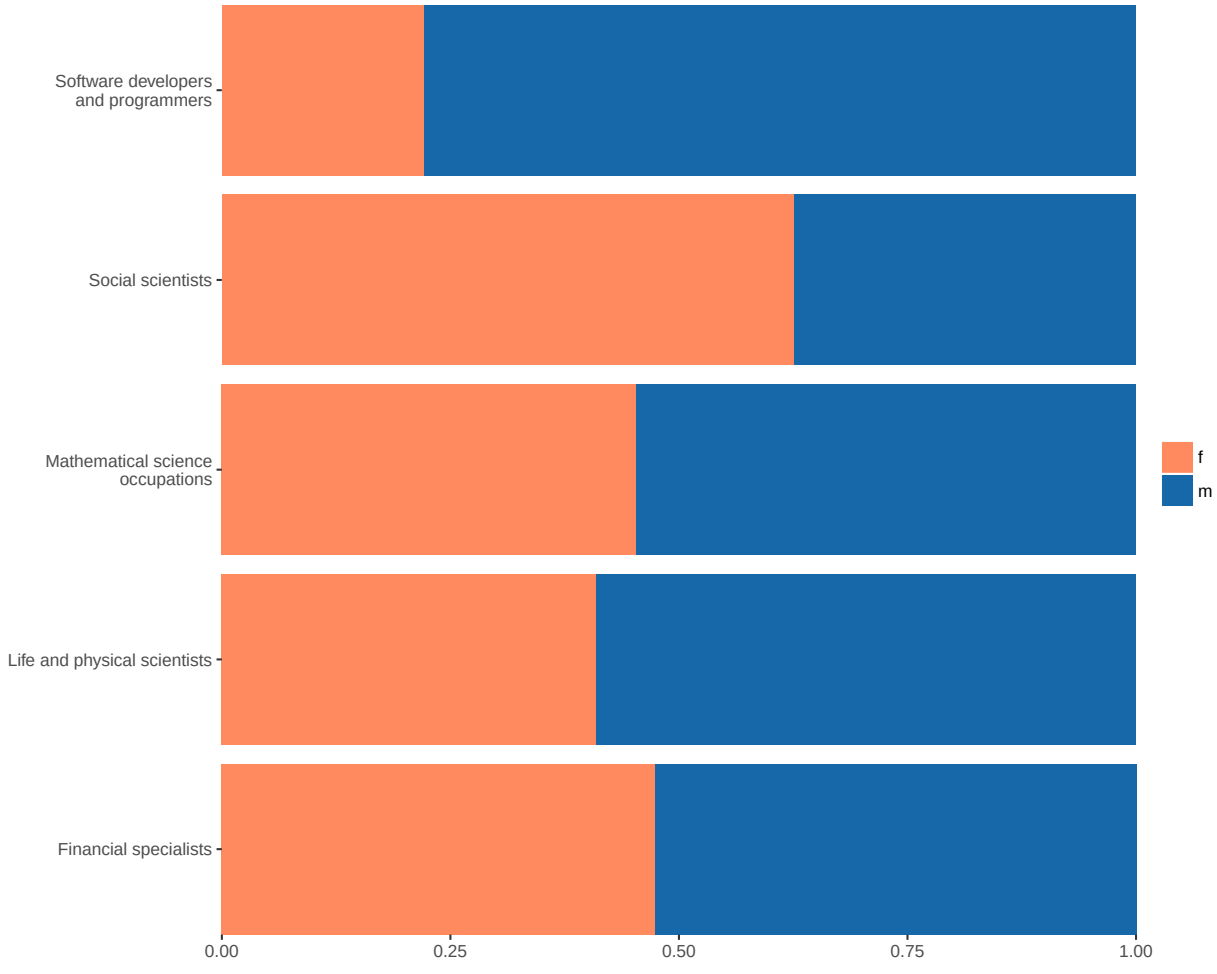
Updating ggplots

```
p
```



Updating ggplots

```
us_census <- import("us_census.txt")
p %+% us_census
```



Updating ggplots

For the data replacement to work, the new data must contain equivalent columns with the same names: here we require `Group`, `Gender`, `Percentage`

```
head(us_census, 4)
```

```
##                   Group Gender   Freq Margin of error Percentage
## 1 Financial specialists      f 656609            11416      0.474
## 2 Financial specialists      m 729150            11629      0.526
## 3     Social scientists      f 185321             7025      0.627
## 4     Social scientists      m 110445             4848      0.373
```

When running similar analyses on different data sets, it can make sense to rename variables as necessary to have common names across the data sets.

Changes to the theme, including titles for the legend, axis etc, can be added, e.g.

```
p %+% us_census +
    ylab("Percentage")
```

Chunk Reuse

Code chunks in rmarkdown and knitr documents can be referenced to re-run the code at another point in the document, e.g.

````
```{r import}
library(dplyr)
car <- import("car_income.txt")
car <- as_data_frame(car)
```


```{r model}
mod <- glm(purchase ~ income + age, family = "binomial", data = car)
round(coef(summary(mod)), 2)
```


```{r age-factor}
car <- mutate(car, age = factor(age))
```


```{r model-factor, ref.label = "model"}
```
````

Chunk Reuse

```
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)    -4.74       2.10   -2.25     0.02
## income          0.07       0.03    2.41     0.02
## age             0.60       0.39    1.53     0.12
```

```
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)    -5.65       2.65   -2.13     0.03
## income          0.09       0.04    2.14     0.03
## age2            0.99       1.59    0.62     0.53
## age3            1.80       1.83    0.98     0.33
## age4            4.28       2.24    1.91     0.06
## age5            3.26       2.06    1.58     0.11
## age6          -12.84    2399.55   -0.01     1.00
```

Custom Functions

The most general way to repeat a task is to write a function to perform that task.

This has several advantages over the previous approaches

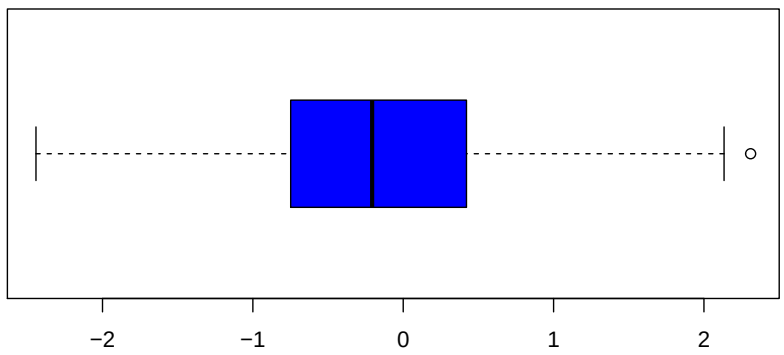
- can be used for any task you can write R code for
- can be used to run the task the first time
- can define arguments to change any aspect of the R code
- can externalize in script or package to use in different analyses/reports

Function Structure

A function has two main components, the **arguments** (inputs to the function) and the **body** (code implementing actions of function)

For example, the following takes the arguments `x` and `col` and plots a boxplot

```
horizBoxplot <- function(x, col){
    boxplot(x, col = col, horizontal = TRUE)
}
horizBoxplot(rnorm(100), col = "blue")
```



Specified Arguments

```
horizBoxplot <- function(x, col = "blue"){
    boxplot(x, col = col, horizontal = TRUE)
}
```

`x` and `col` are specified arguments. The user can pass objects to these arguments using their names or by supplying unnamed in the right order

```
horizBoxplot(col = "red", x = rnorm(100))
horizBoxplot(rnorm(100), "red")
```

`col` has the default value `"blue"` which will be used if the user does not pass a value to this argument. `x` has no default value so the function fails if this argument is not specified

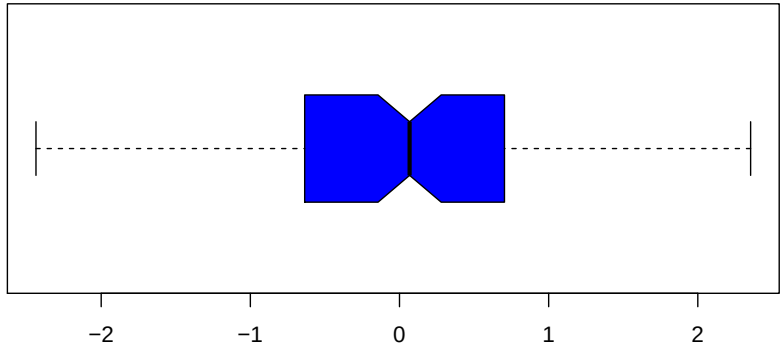
```
horizBoxplot(col = "red")


## Error in boxplot(x, col = col, horizontal = TRUE): argument "x" is
missing, with no default
```

Unspecified Arguments

```
horizBoxplot <- function(x, col = "blue", ...){
    boxplot(x, col = col, horizontal = TRUE, ...)
}
```

`...` allows unspecified arguments to be passed to the function. This device is used by functions that work with arbitrary numbers of objects, e.g. `sum`. Here it is used to pass on further arguments to `boxplot`, e.g.

```
horizBoxplot(rnorm(100), notch = TRUE)
```



Return Values and Side-effects

The display of the boxplot is a side-effect: a change outside the function that occurs when the function is run. Other examples include writing to files or printing output.

Functions also return a value, by default, this is the object created by the last line of code, e.g.

```
res <- horizBoxplot(rnorm(100), col = "blue")
str(res)
```

```
## List of 6
##  $ stats: num [1:5, 1] -2.035 -0.4 0.164 0.766 2.437
##  $ n    : num 100
##  $ conf : num [1:2, 1] -0.0203 0.3481
##  $ out  : num [1:2] -2.6 -2.19
##  $ group: num [1:2] 1 1
##  $ names: chr "1"
```

Custom ggplot

Suppose we wish to convert the following code for a horizontal ggplot to a function

```
ggplot(car, aes(x = factor(1), y = income)) +
    geom_boxplot() + coord_flip() +
    theme(axis.title.y = element_blank(),
          axis.text.y = element_blank(),
          axis.ticks.y = element_blank())
```

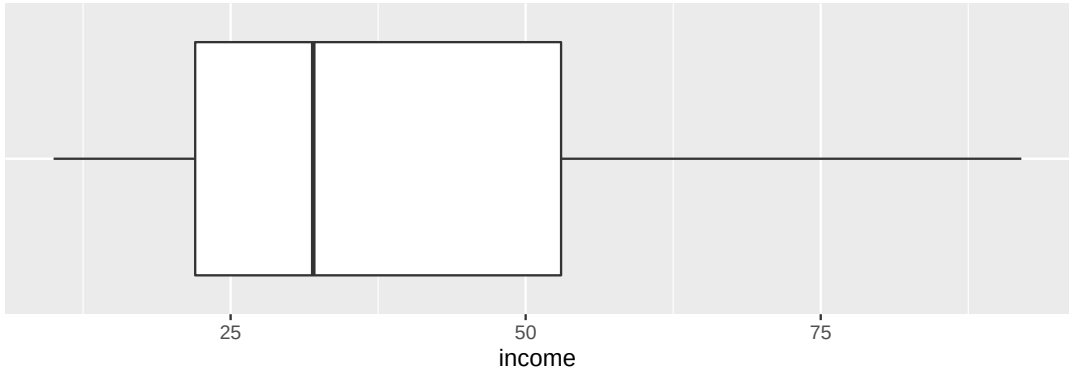
In interactive use, we can specify aesthetic mappings to `aes` using expressions involving variable names.

To define the mappings using function arguments, we can pass strings to `aes_string`

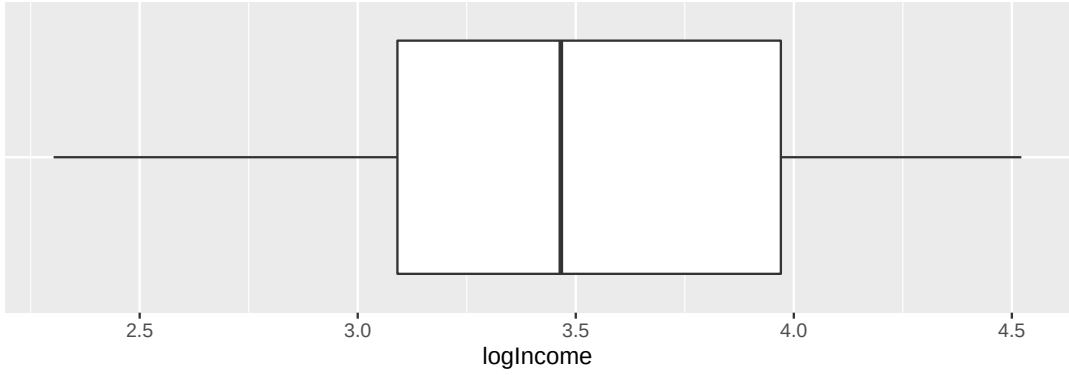
```
ggHorizBoxplot <- function(data, string){
    data$x <- factor(1)
    ggplot(data, aes_string(x = "x", y = string)) +
        geom_boxplot() + coord_flip() +
        theme(axis.title.y = element_blank(),
              axis.text.y = element_blank(),
              axis.ticks.y = element_blank())
}
```

Custom ggplot

```
ggHorizBoxplot(car, "income")
```




```
car <- mutate(car, logIncome = log(income))
ggHorizBoxplot(car, "logIncome")
```



Custom Data Processing

The following code finds the frequencies for each level of the age factor

```
car %>%
    group_by(age) %>%
    summarize(Freq = n())
```

```
## # A tibble: 6 × 2
##      age  Freq
##   <fctr> <int>
## 1      1     4
## 2      2     8
## 3      3    11
## 4      4     5
## 5      5     4
## 6      6     1
```

Custom Data Processing

Like ggplot2, **dplyr** functions use expressions of variables that are evaluated in a data frame.

The demonstrates the "rename trick" to get around this

```
freqTable <- function(data, string){
    data %>%
        rename_(x = string) %>% # note underscore in `rename_`
        group_by(x) %>%
        summarize(Freq = n())
}
freqTable(car, "age")
```

```
## # A tibble: 6 × 2
##         x  Freq
##    <fctr> <int>
## 1      1     4
## 2      2     8
## 3      3    11
## 4      4     5
## 5      5     4
## 6      6     1
```

Converting Code to a Function

A good rule of thumb is if you find yourself copy-pasting a chunk of code more than two/three times, convert it to a function.

In RStudio, select the code to convert, go to the *Code* menu and choose *Extract Function*. The code will be wrapped in a function and objects undefined by the code will be turned in arguments

```
extractedFunction <- function(car, income) {
 ggplot(car, aes(x = factor(1), y = income)) +
     geom_boxplot() + coord_flip() +
     theme(axis.title.y = element_blank(),
           axis.text.y = element_blank(),
           axis.ticks.y = element_blank())
}
```

Both arguments and code may need editing to get a working function!

Externalizing Function Code

To declutter your script and enable the functions to be used in other scripts, put functions in a separate `.R` file and source in, e.g.

```
source("customFunctions.R")
```

Even for short functions it is worth adding some comments to your code so that you remember what it does

```
# get group frequencies
# data: a data frame containing the variable
# var: a string specifying the grouping variable
groupFreq <- function(data, var){
    data %>%
        rename_(x = var) %>% # rename trick!
        group_by_(x) %>%
        summarize(Freq = n())
}
```

Iteration

Wrapping up code in a function means that repeating a similar action requires only a line or two of new code. However if we want to use the function many times, we're back to the problems of copy-paste.

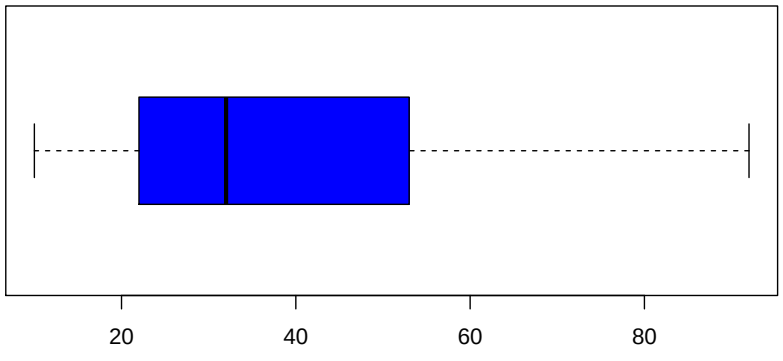
The **purrr** package (with three 'r's!) provides functions to map objects to function arguments, so the function can be run with different arguments and the results returned in a convenient form.

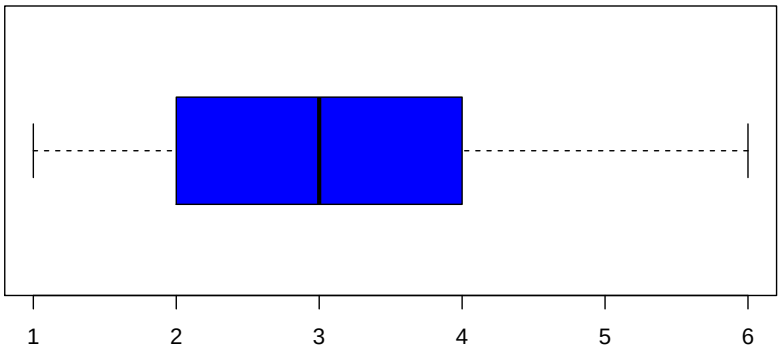
Iteration on One Argument for Side Effect

Suppose we want to produce a horizontal boxplot for multiple variables in a data frame.

Since we are only interested in the side-effect (plot) we use the `walk` function. This "walks" along the elements of a vector or list, using each element in turn as the *first* argument to our function.

```
library(purrr)
car <- mutate(car, age = as.numeric(age))
walk(select(car, -purchase), horizBoxplot)
```

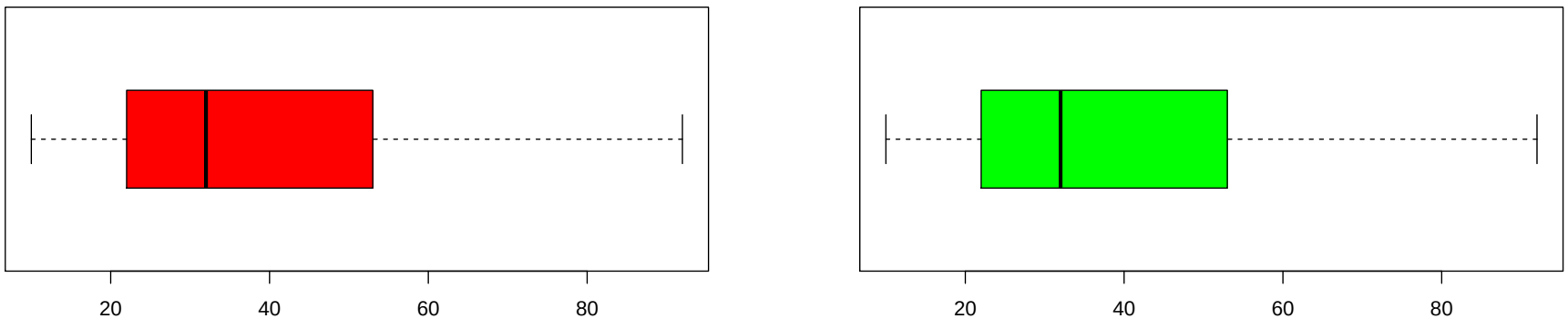




Iteration on *n*'th Argument

To iterate over an argument other than the first, we can specify all preceding arguments

```
walk(c("red", "green"),
     horizBoxplot, x = car$income)
```



Alternatively we can define an *anonymous* function on the fly

```
walk(c("red", "green"),
     function(el) horizBoxplot(car$income, col = el))
```

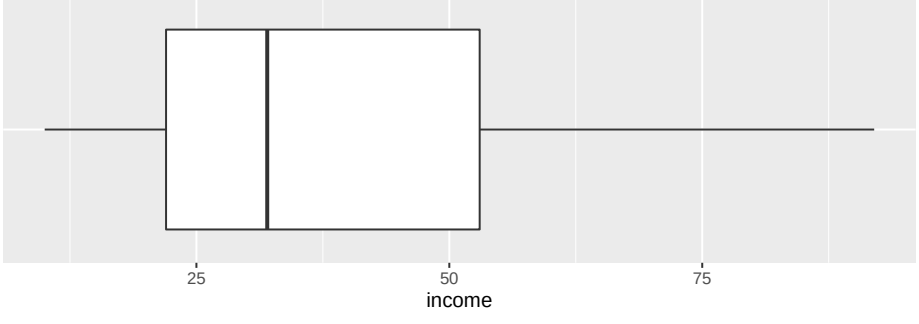
Another alternative is to use `pwalk` as shown later.

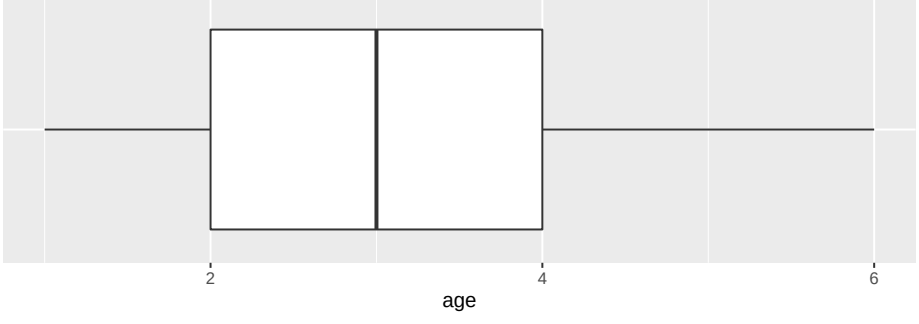
Iterating on One Argument for Return Value

A `ggplot` only displays when the object is printed. Therefore to display the result of `ggHorizPlot` for multiple variables we must collect the return value and then print.

`map` is equivalent to `walk`, but collects the return values in a list

```
res <- map(c(income = "income", age = "age"),
           ggHorizBoxplot, data = car)
res$income
res$age
```





Iterating on One Argument for Return Value

Often we will want to combine the results in a more convenient data structure.

For example, we can read in multiple data files and combine in a single data frame using `map_df`

```
football <- map_df(c(`2008-9` = "2008-9.tsv", `2009-10` = "2009-10.tsv"),
                   import, .id = "Season")
head(football, 2)
```

```
##   Season Home Away Home Score Away Score
## 1 2008-9  Sto  Hul          1          1
## 2 2008-9  Wig  Hul          1          0
```

The functions `map_lgl`, `map_chr`, `map_int`, `map_dbl`, work in a similar way to combine results in a logical/character/integer/numeric vector.

Iterating on One Argument for Return Value

The examples so far have iterated over the elements of a vector or the vectors in a data frame (a special case of a list).

To iterate over other objects, rather than create a (large) list of the objects it is better to iterate over their names, using `get` to get each object as necessary

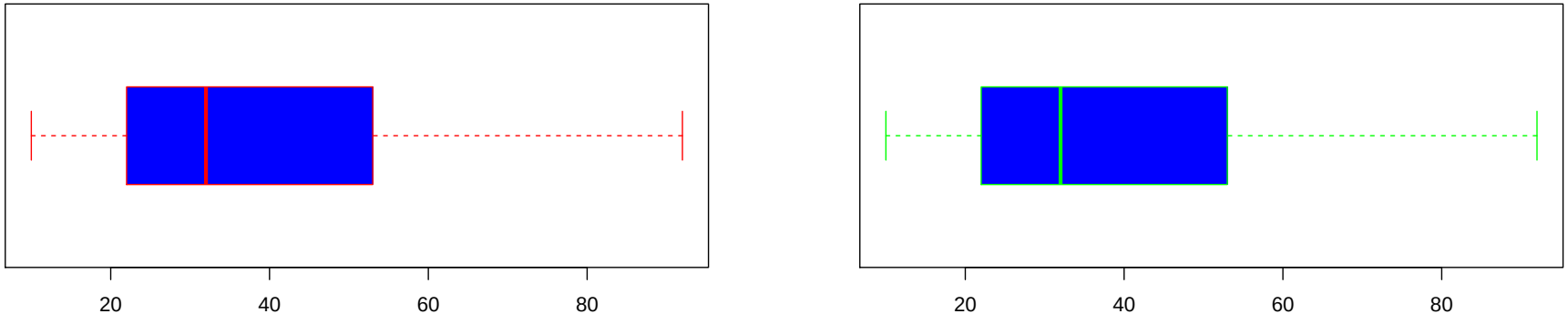
```
datasets <- c("r_community", "us_census")
nobs <- map_int(datasets, function(x) nrow(get(x)))
nobs


## [1] 14 10
```

Iterating on Multiple Arguments

The `walk` and `map*` functions all have `p*` equivalents to iterate on multiple arguments in parallel, e.g. `pwalk`, `pmap`.

All the `p*` functions iterate over a list, where each element corresponds to a function argument to iterate on. This gives another way to iterate on the n'th argument: by name, e.g.

```
pwalk(list(border = c("red", "green")),
      horizBoxplot, x = car$income)
```

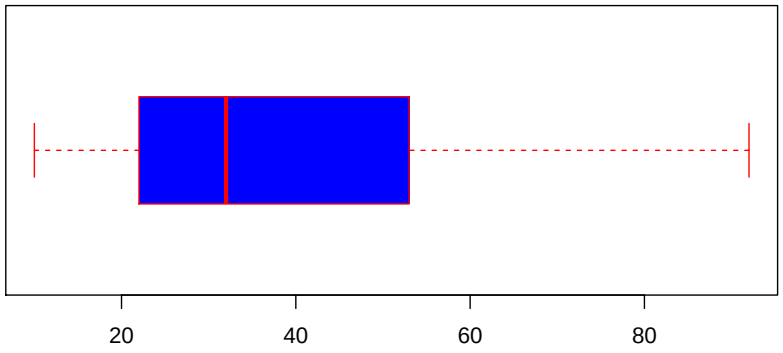


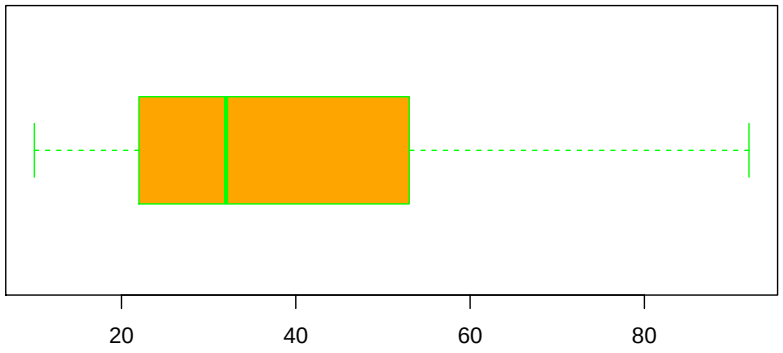
Only compulsory additional arguments need be specified.

Iterating on Multiple Arguments

The extension to multiple arguments is straight-forward:

```
pwalk(list(col = c("blue", "orange"),
           border = c("red", "green")),
      horizBoxplot, x = car$income)
```





Iterating on Multiple Arguments

The `cross_d` from **purrr** is useful when we want to iterate over all combinations of variables

```
fac <- list(patient = c("A", "B", "C"),
            sample = c("Blood", "Synovium"))
fac <- cross_d(fac)
fac
```

```
## # A tibble: 6 × 2
##   patient   sample
##     <chr>    <chr>
## 1       A    Blood
## 2       B    Blood
## 3       C    Blood
## 4       A Synovium
## 5       B Synovium
## 6       C Synovium
```

Iterating on Multiple Parameters

The crossed variables can then be mapped to function arguments *with the same name*

```
readData <- function(patient, sample){
    nm <- paste0("Patient_", patient, "_", sample, "_TCRB.tsv")
    as_data_frame(import(nm))
}
TCRB <- pmap_df(fac, readData, .id = "Group")
id <- as.numeric(TCRB$Group)
TCRB <- bind_cols(slice(fac, id), TCRB)
head(TCRB, 2)
```

```
## # A tibble: 2 × 8
##   patient sample Group count frequencyCount cdr3Length
##     <chr>  <chr> <chr> <int>          <dbl>      <int>
## 1       A  Blood     1   320         0.0247         48
## 2       A  Blood     1   319         0.0247         42
## # ... with 2 more variables: nucleotide <chr>, aminoAcid <chr>
```