

R Basics

Fundamental Techniques in Data Science



**Utrecht
University**

Kyle M. Lang

Department of Methodology & Statistics
Utrecht University

Outline

Functions

Iteration

Data Manipulation

- Subsetting

- Transforming & Rearranging

Pipes

- The Basic Pipe: `|>`

- Other Flavors of Pipe

Workflow & Project Management



Attribution

This course was originally developed by Gerko Vink. You can access the original version of these materials on Dr. Vink's GitHub page:

<https://github.com/gerkovink/fundamentals>.

The course materials have been (extensively) modified. Any errors or inaccuracies introduced via these modifications are fully my own responsibility and shall not be taken as representing the views and/or beliefs of Dr. Vink.

You can see Gerko's version of the course on his personal website:

<https://www.gerkovink.com/fundamentals>.



Prerequisite Knowledge

After completing the preparatory exercises, you should already be familiar with the following ideas.

- What is R?
- What is RStudio?
- Basic R data objects
 - Atomic Vectors
 - Matrices
 - Lists
 - Data Frames
 - Factors
- Visualization with Base R graphics

We will not cover these topics in the lectures.



Related Topics

As part of this week's lab exercises, you will complete detailed tutorials covering:

- How to create reproducible reports with Quarto
- How to load data from external files

We will not cover these topics in the lectures.



FUNCTIONS



R Functions

Functions are the foundation of R programming.

- Other than data objects, almost everything else that you interact with when using R is a function.
- Any R command written as a word followed by parentheses, `()`, is a function.
 - `mean()`
 - `library()`
 - `mutate()`
- Infix operators are aliased functions.
 - `<-`
 - `+`, `-`, `*`
 - `>`, `<`, `==`



User-Defined Functions

We can define our own functions using the `function()` function.

```
square <- function(x) {  
  out <- x^2  
  out  
}
```

After defining a function, we call it in the usual way.

```
square(5)
```

```
[1] 25
```

One-line functions don't need braces.

```
square <- function(x) x^2
```

```
square(5)
```

```
[1] 25
```


User-Defined Functions

Function arguments are not strictly typed.

```
square(1:5)
```

```
[1] 1 4 9 16 25
```

```
square(pi)
```

```
[1] 9.869604
```

```
square(TRUE)
```

```
[1] 1
```

But there are limits.

```
square("bob") # But one can only try so hard
```

```
Error in x^2: non-numeric argument to binary operator
```

User-Defined Functions

Functions can take multiple arguments.

```
mod <- function(x, y) x %% y
mod(10, 3)

[1] 1
```

Sometimes it's useful to specify a list of arguments.

```
getLsBeta <- function(datList) {
  X <- datList$X
  y <- datList$y

  solve(crossprod(X)) %*% t(X) %*% y
}
```

User-Defined Functions

```
X      <- matrix(runif(500), ncol = 5)
datList <- list(y = X %*% rep(0.5, 5), X = X)

getLsBeta(datList = datList)
```

```
      [,1]
[1,] 0.5
[2,] 0.5
[3,] 0.5
[4,] 0.5
[5,] 0.5
```

User-Defined Functions

Functions are first-class objects in R.

- We can treat functions like any other R object.

R views an unevaluated function as an object with type "closure".

```
class(getLsBeta)
[1] "function"

typeof(getLsBeta)
[1] "closure"
```

An evaluated functions is equivalent to the objects it returns.

```
class(getLsBeta(datList))
[1] "matrix" "array"

typeof(getLsBeta(datList))
[1] "double"
```

Nested Functions

We can use functions as arguments to other operations and functions.

```
fun1 <- function(x, y) x + y

## What will this command return?
fun1(1, fun1(1, 1))

[1] 3
```

Why would we care?

```
s2 <- var(runif(100))
x <- rnorm(100, 0, sqrt(s2))
```

Nested Functions

```
X[1:8, ]
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	0.52431382	0.67136447	0.28228726	0.7148383	0.54204681
[2,]	0.01926742	0.11693762	0.09148502	0.6929171	0.88371944
[3,]	0.05100735	0.18432074	0.43547799	0.6097462	0.09026598
[4,]	0.60566972	0.12944127	0.21000143	0.2441917	0.68141473
[5,]	0.48737303	0.94030405	0.23988619	0.4915910	0.36353771
[6,]	0.19941958	0.96670678	0.11455820	0.1243947	0.24253273
[7,]	0.95507804	0.38705829	0.49733535	0.2968470	0.81001800
[8,]	0.11093197	0.07731757	0.84923006	0.8653987	0.61914193

```
c(1, 3, 6:9, 12)
```

```
[1] 1 3 6 7 8 9 12
```

ITERATION



Loops

There are three types of loops in R: *for*, *while*, and *until*.

- You'll rarely use anything but the *for* loop.
- So, we won't discuss *while* or *until* loops.

A *for loop* is defined as follows.

```
for(INDEX in RANGE) { Stuff To Do with the Current INDEX Value }
```



Loops

For example, the following loop will sum the numbers from 1 to 100.

```
val <- 0
for(i in 1:100) {
  val <- val + i
}
```

```
val
```

```
[1] 5050
```

Loops

This loop will compute the mean of every column in the `mtcars` data.

```
means <- rep(0, ncol(mtcars))
for(j in 1:ncol(mtcars)) {
  means[j] <- mean(mtcars[, j])
}
```

means

```
[1] 20.090625  6.187500 230.721875 146.687500  3.596563
[6]  3.217250 17.848750  0.437500  0.406250  3.687500
[11]  2.812500
```

Loops

Loops are often one of the least efficient solutions in R.

```
n <- 1e8

t0 <- system.time({
  val0 <- 0
  for(i in 1:n) val0 <- val0 + i
})

t1 <- system.time(
  val1 <- sum(1:n)
)
```

Loops

Both approaches produce the same answer.

```
val0 - val1
```

```
[1] 0
```

But the loop is many times slower.

```
t0
```

user	system	elapsed
1.267	0.000	1.267

```
t1
```

user	system	elapsed
0	0	0

Loops

There is often a built in routine for what you are trying to accomplish with the loop.

```
## The appropriate way to get variable means:
```

```
colMeans(mtcars)
```

mpg	cyl	disp	hp	drat
20.090625	6.187500	230.721875	146.687500	3.596563
wt	qsec	vs	am	gear
3.217250	17.848750	0.437500	0.406250	3.687500
carb				
2.812500				

Apply Statements

In R, some flavor of *apply statement* is often preferred to a loop.

- Apply statements broadcast some operation across the elements of a data object.
- Apply statements can take advantage of internal optimizations that loops can't use.

There are many flavors of apply statement in R, but the three most common are:

- `apply()`
- `lapply()`
- `sapply()`



Apply Statements

Apply statements generally take one of two forms:

```
apply(DATA, MARGIN, FUNCTION, ...)
```

```
apply(DATA, FUNCTION, ...)
```



Apply Examples

```
## Load some example data:
```

```
data(mtcars)
```

```
## Subset the data:
```

```
dat1 <- mtcars[1:5, 1:3]
```

```
## Find the range of each row:
```

```
apply(dat1, 1, range)
```

	Mazda RX4	Mazda RX4 Wag	Datsun 710	Hornet 4 Drive
[1,]	6	6	4	6
[2,]	160	160	108	258

	Hornet Sportabout
[1,]	8
[2,]	360

Apply Examples

```
## Find the maximum value in each column:
```

```
apply(dat1, 2, max)
```

```
   mpg   cyl  disp  
22.8   8.0 360.0
```

```
## Subtract 1 from every cell:
```

```
apply(dat1, 1:2, function(x) x - 1)
```

	mpg	cyl	disp
Mazda RX4	20.0	5	159
Mazda RX4 Wag	20.0	5	159
Datsun 710	21.8	3	107
Hornet 4 Drive	20.4	5	257
Hornet Sportabout	17.7	7	359

Apply Examples

```
## Create a toy list:
l1 <- list()
for(i in 1:3) l1[[i]] <- runif(10)

## Find the mean of each list entry:
lapply(l1, mean)

[[1]]
[1] 0.526697

[[2]]
[1] 0.4020885

[[3]]
[1] 0.607818

## Same as above, but return the result as a vector:
sapply(l1, mean)

[1] 0.5266970 0.4020885 0.6078180
```

Apply Examples

```
## Find the range of each list entry:
```

```
lapply(l1, range)
```

```
[[1]]
```

```
[1] 0.04395916 0.99350611
```

```
[[2]]
```

```
[1] 0.002797563 0.821082495
```

```
[[3]]
```

```
[1] 0.09926892 0.90430843
```

```
sapply(l1, range)
```

```
      [,1]      [,2]      [,3]
```

```
[1,] 0.04395916 0.002797563 0.09926892
```

```
[2,] 0.99350611 0.821082495 0.90430843
```

Apply Examples

We can add additional arguments needed by the function.

- These arguments must be named.

```
apply(dat1, 2, mean, trim = 0.1)
```

mpg	cyl	disp
20.98	6.00	209.20

```
sapply(dat1, mean, trim = 0.1)
```

mpg	cyl	disp
20.98	6.00	209.20

DATA MANIPULATION



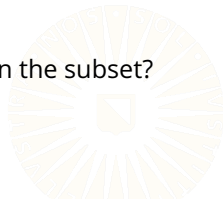
Base R Subsetting

In Base R, we typically use three operators to subset objects:

- `[]`
- `[[]]`
- `$`

Which of these operators we choose to use (and how we implement the chosen operator) will depend on two criteria:

- What type of object are we trying to subset?
- How much of the original typing do we want to keep in the subset?



Some Data

Let's create a few data frames to work with in the next few slides.

```
(d1 <- data.frame(1:6, c(-1, 1), seq(0.1, 0.6, 0.1)))
```

	X1.6	c..1..1.	seq.0.1..0.6..0.1.
1	1	-1	0.1
2	2	1	0.2
3	3	-1	0.3
4	4	1	0.4
5	5	-1	0.5
6	6	1	0.6

Some Data

```
(d2 <- data.frame(x = 1:6, y = c(-1, 1), z = seq(0.1, 0.6, 0.1)))
```

	x	y	z
1	1	-1	0.1
2	2	1	0.2
3	3	-1	0.3
4	4	1	0.4
5	5	-1	0.5
6	6	1	0.6

Some Data

```
(d3 <- data.frame(a = sample(c(TRUE, FALSE), 8, replace = TRUE),  
                 b = sample(c("foo", "bar"), 8, replace = TRUE),  
                 c = runif(8))  
)
```

	a	b	c
1	TRUE	bar	0.9761304
2	FALSE	foo	0.3529488
3	FALSE	bar	0.4843390
4	TRUE	foo	0.4816117
5	TRUE	bar	0.3965226
6	TRUE	bar	0.3782633
7	TRUE	bar	0.3213372
8	TRUE	bar	0.8692548

Some Data

```
(d4 <- data.frame(matrix(NA, 10, 3)))
```

	X1	X2	X3
1	NA	NA	NA
2	NA	NA	NA
3	NA	NA	NA
4	NA	NA	NA
5	NA	NA	NA
6	NA	NA	NA
7	NA	NA	NA
8	NA	NA	NA
9	NA	NA	NA
10	NA	NA	NA

Tidyverse Subsetting

The **dplyr** package provides many ways to subset data, but two functions are most frequently useful.

- `select()` : subset columns
- `filter()` : subset rows

```
library(dplyr)
```

Subsetting Columns: `select()`

The `dplyr::select()` function provides a very intuitive syntax for variable selection and column-wise subsetting.

```
select(d3, a, b)
```

	a	b
1	TRUE	bar
2	FALSE	foo
3	FALSE	bar
4	TRUE	foo
5	TRUE	bar
6	TRUE	bar
7	TRUE	bar
8	TRUE	bar

```
select(d3, -a)
```

	b	c
1	bar	0.9761304
2	foo	0.3529488
3	bar	0.4843390
4	foo	0.4816117
5	bar	0.3965226
6	bar	0.3782633
7	bar	0.3213372
8	bar	0.8692548

Subsetting Rows

The `dplyr::filter()` function provides easy row subsetting:

```
filter(d3, c > 0.5)
```

	a	b	c
1	TRUE	bar	0.9761304
2	TRUE	bar	0.8692548

```
filter(d3, c > 0.15, b == "foo")
```

	a	b	c
1	FALSE	foo	0.3529488
2	TRUE	foo	0.4816117

We can achieve the same effect via logical indexing in Base R:

```
d3[d3$c > 0.5, ]
```

	a	b	c
1	TRUE	bar	0.9761304
8	TRUE	bar	0.8692548

```
d3[d3$c > 0.15 & d3$b == "foo", ]
```

	a	b	c
2	FALSE	foo	0.3529488
4	TRUE	foo	0.4816117

Base R Variable Transformations

There is nothing very special about the process of transforming variables in Base R.

```
d4 <- d3
d4$d <- scale(d4$c)
d4$e <- !d4$a
d4
```

	a	b	c	d	e
1	TRUE	bar	0.9761304	1.7809902	FALSE
2	FALSE	foo	0.3529488	-0.7211104	TRUE
3	FALSE	bar	0.4843390	-0.1935730	TRUE
4	TRUE	foo	0.4816117	-0.2045233	FALSE
5	TRUE	bar	0.3965226	-0.5461597	FALSE
6	TRUE	bar	0.3782633	-0.6194716	FALSE
7	TRUE	bar	0.3213372	-0.8480321	FALSE
8	TRUE	bar	0.8692548	1.3518800	FALSE

```
d4 <- d3
d4$c <- scale(d4$c, scale = FALSE)
d4$a <- as.numeric(d4$a)
d4
```

	a	b	c
1	1	bar	0.44357942
2	0	foo	-0.17960218
3	0	bar	-0.04821196
4	1	foo	-0.05093925
5	1	bar	-0.13602839
6	1	bar	-0.15428768
7	1	bar	-0.21121374
8	1	bar	0.33670378

Tidyverse Variable Transformations

The `mutate()` function from **dplyr** is the workhorse of Tidyverse transformation functions.

```
mutate(d3, d = rbinom(nrow(d3), 1, c))
```

	a	b	c	d
1	TRUE	bar	0.9761304	1
2	FALSE	foo	0.3529488	0
3	FALSE	bar	0.4843390	0
4	TRUE	foo	0.4816117	1
5	TRUE	bar	0.3965226	1
6	TRUE	bar	0.3782633	0
7	TRUE	bar	0.3213372	0
8	TRUE	bar	0.8692548	1

```
mutate(d3,  
  d = rbinom(nrow(d3), 1, c),  
  e = d * c  
)
```

	a	b	c	d	e
1	TRUE	bar	0.9761304	1	0.9761304
2	FALSE	foo	0.3529488	1	0.3529488
3	FALSE	bar	0.4843390	0	0.0000000
4	TRUE	foo	0.4816117	1	0.4816117
5	TRUE	bar	0.3965226	0	0.0000000
6	TRUE	bar	0.3782633	1	0.3782633
7	TRUE	bar	0.3213372	0	0.0000000
8	TRUE	bar	0.8692548	0	0.0000000

Sorting & Ordering

To sort a single vector, the best option is the Base R `sort()` function.

```
sort(d3$c)
```

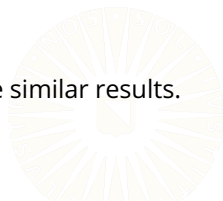
```
[1] 0.3213372 0.3529488 0.3782633 0.3965226 0.4816117  
[6] 0.4843390 0.8692548 0.9761304
```

```
sort(d3$c, decreasing = TRUE)
```

```
[1] 0.9761304 0.8692548 0.4843390 0.4816117 0.3965226  
[6] 0.3782633 0.3529488 0.3213372
```

To sort the rows of a data frame according to the order of one of its columns, the `dplyr::arrange()` works best.

- You can use the Base R `order()` function to achieve similar results.
- The behavior of `order()` is (extremely) unintuitive.



Tidyverse Ordering

Using `dplyr::arrange()` could not be simpler.

```
arrange(d3, a)
```

	a	b	c
1	FALSE	foo	0.3529488
2	FALSE	bar	0.4843390
3	TRUE	bar	0.9761304
4	TRUE	foo	0.4816117
5	TRUE	bar	0.3965226
6	TRUE	bar	0.3782633
7	TRUE	bar	0.3213372
8	TRUE	bar	0.8692548

```
arrange(d3, -c)
```

	a	b	c
1	TRUE	bar	0.9761304
2	TRUE	bar	0.8692548
3	FALSE	bar	0.4843390
4	TRUE	foo	0.4816117
5	TRUE	bar	0.3965226
6	TRUE	bar	0.3782633
7	FALSE	foo	0.3529488
8	TRUE	bar	0.3213372

```
arrange(d3, -a, c)
```

	a	b	c
1	TRUE	bar	0.3213372
2	TRUE	bar	0.3782633
3	TRUE	bar	0.3965226
4	TRUE	foo	0.4816117
5	TRUE	bar	0.8692548
6	TRUE	bar	0.9761304
7	FALSE	foo	0.3529488
8	FALSE	bar	0.4843390

PIPES



What are pipes?

The `|>` symbol represents the *pipe* operator.

- We use the pipe operator to compose functions into a *pipeline*.

The following code represents a pipeline.

```
firstBoys <-  
  here::here("data", "boys.rds") |>  
  readRDS() |>  
  head()
```

This pipeline replaces the following code.

```
firstBoys <- head(readRDS(here::here("data", "boys.rds")))
```

Why are pipes useful?

Let's assume that we want to:

1. Load data
2. Transform a variable
3. Filter cases
4. Select columns

Without a pipe, we may do something like this:

```
library(dplyr)

boys <- readRDS(here::here("data", "boys.rds"))
boys <- transform(boys, hgt = hgt / 100)
boys <- filter(boys, age > 15)
boys <- subset(boys, select = c(hgt, wgt, bmi))
```

Why are pipes useful?

With the pipe, we could do something like this:

```
boys <-  
  readRDS(here::here("data", "boys.rds")) |>  
  transform(hgt = hgt / 100) |>  
  filter(age > 15) |>  
  subset(select = c(hgt, wgt, bmi))  
  
Error in parse(text = input):  <text>:6:0:  unexpected end of input  
4:      filter(age > 15) |>  
5:      subset(select = c(hgt, wgt, bmi))  
  ^
```

With a pipeline, our code more clearly represents the sequence of steps in our analysis.

Benefits of Pipes

When you use pipes, your code becomes more readable.

- Operations are structured from left to right instead of in to out.
- You can avoid many nested function calls.
- You don't have to keep track of intermediate objects.
- It's easy to add steps to the sequence.

In RStudio, you can use a keyboard shortcut to insert the `|>` symbol.

- Windows/Linux: *ctrl + shift + m*
- Mac: *cmd + shift + m*



What do pipes do?

Pipes compose R functions without nesting.

- `f(x)` becomes `x |> f()`

```
mean(rnorm(10))
```

```
[1] -0.4146355
```

```
rnorm(10) |> mean()
```

```
[1] -0.02220906
```

What do pipes do?

Multiple function arguments are fine.

- `f(x, y)` becomes `x |> f(y)`

```
cor(boys, use = "pairwise.complete.obs")
```

	hgt	wgt	bmi
hgt	1.0000000	0.6100784	0.1758781
wgt	0.6100784	1.0000000	0.8841304
bmi	0.1758781	0.8841304	1.0000000

```
boys |> cor(use = "pairwise.complete.obs")
```

	hgt	wgt	bmi
hgt	1.0000000	0.6100784	0.1758781
wgt	0.6100784	1.0000000	0.8841304
bmi	0.1758781	0.8841304	1.0000000

What do pipes do?

Composing more than two functions is easy, too.

- `h(g(f(x)))` becomes `x |> f() |> g() |> h()`

```
max(na.omit(subset(boys, select = wgt)))
```

```
[1] 117.4
```

```
boys |>
```

```
  subset(select = wgt) |>
```

```
  na.omit() |>
```

```
  max()
```

```
[1] 117.4
```

The Role of `_` in a Pipeline

In the expression `a |> f(arg1, arg2, arg3)`, `a` will be "piped into" `f()` as `arg1`.

```
data(cats, package = "MASS")  
cats |> plot(Hwt ~ Bwt)
```

```
Error in text.default(x, y, txt, cex = cex, font = font): invalid  
mathematical annotation
```

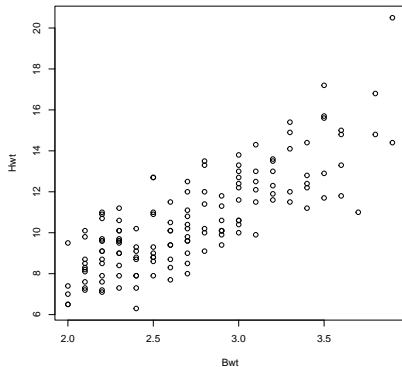
Clearly, we have a problem if we pipe our data into the wrong argument.

- We can change this behavior with the `_` symbol.
- The `_` symbol acts as a placeholder for the data in a pipeline.



The Role of `_` in a Pipeline

```
cats |> plot(Hwt ~ Bwt, data = _)
```

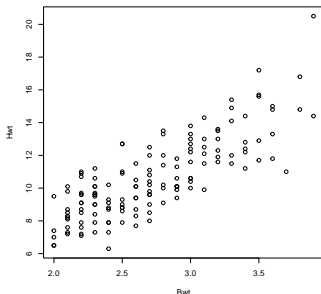


Exposition Pipe: %\$%

The **magrittr** package provides several different flavors of pipe. The *exposition pipe*, `%$%`, is a particularly useful variant.

- The exposition pipe *exposes* the contents of an object to the next function in the pipeline.

```
library(magrittr)
cats %$% plot(Hwt ~ Bwt)
```



Performing a T-Test in a Pipeline

```
cats %>% t.test(Hwt ~ Sex)
```

Welch Two Sample t-test

data: Hwt by Sex

t = -6.5179, df = 140.61, p-value = 1.186e-09

alternative hypothesis: true difference in means between group F and group M
is not equal to 0

95 percent confidence interval:

-2.763753 -1.477352

sample estimates:

mean in group F mean in group M

9.202128

11.322680

The above is equivalent to either of the following.

```
cats |> t.test(Hwt ~ Sex, data = _)
```

```
t.test(Hwt ~ Sex, data = cats)
```

WORKflow & PROJECT MANAGEMENT



Some Programming Tips

You can save yourself a great deal of heartache by following a few simple guidelines.

- Keep your code tidy.
- Use comments to clarify what you are doing.
- When working with functions in RStudio, use the TAB key to quickly access the documentation of the function's arguments.
- Give your R scripts and objects meaningful names.
- Use a consistent directory structure and RStudio projects.



General Style Advice

Use common sense and BE CONSISTENT.

- Browse the [tidyverse style guide](#).
 - The point of style guidelines is to enforce a common vocabulary.
 - You want people to concentrate on *what* you're saying, not *how* you're saying it.
- If the code you add to a project/codebase looks drastically different from the extant code, the incongruity will confuse readers and collaborators.

Spacing and whitespace are your friends.

- `a<-c(1,2,3,4,5)`
- `a <- c(1, 2, 3, 4, 5)`
- At least put spaces around assignment operators and after every comma!

