MONASH University

MONASH BUSINESS SCHOOL

ETC4500/ETC5450
Advanced R programming

Week 2: Foundations of R programming

`arp.numbat.space`

Outline

Outline

Exercises

1. What is the result of subsetting a vector with positive integers, negative integers, a logical vector, or a character vector?
2. What's the difference between `[`, `[[`, and `$` when applied to a list?
3. When should you use `drop = FALSE`?

Exercises

4. Fix each of the following common data frame subsetting errors:

```
mtcars[mtcars$cyl = 4, ]
mtcars[-1:4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]
```

5. Extract the residual degrees of freedom from `mod`

```
mod <- lm(mpg ~ wt, data = mtcars)
```

6. Extract the R squared from the model summary (`summary(mod)`)

Exercises

7. How would you randomly permute the columns of a data frame?
8. Can you simultaneously permute the rows and columns in one step?
9. How would you select a random sample of m rows from a data frame? What if the sample had to be contiguous (i.e., with an initial row, a final row, and every row in between)?
10. How could you put the columns in a data frame in alphabetical order?

Outline

Exercises

11 What is the difference between `if` and `ifelse()` and `dplyr::if_else()`?

12 What type of vector does each of the following calls to `ifelse()` return?

```
ifelse(TRUE, 1, "no")
ifelse(FALSE, 1, "no")
ifelse(NA, 1, "no")
```

Exercises

13 Why does the following code work?

```
x <- 1:10
if (length(x)) "not empty" else "empty"
```

```
[1] "not empty"
```

```
x <- numeric()
if (length(x)) "not empty" else "empty"
```

```
[1] "empty"
```

Outline

Function fundamentals

- Almost all functions can be broken down into three components: arguments, body, and environment.
  - The `formals()`, the list of arguments that control how you call the function.
  - The `body()`, the code inside the function.
  - The `environment()`, the data structure that determines how the function finds the values associated with the names.
- Functions are objects and have attributes.

Function components

```
f02 <- function(x, y) {
  # A comment
  x + y
}
formals(f02)
```

```
$x

$y
```

```
body(f02)
```

```
{
    x + y
}
```

```
environment(f02)
```

```
<environment: R_GlobalEnv>
```

Function attributes

```
attr(f02, "srcref")
```

```
function(x, y) {
  # A comment
  x + y
}
```

Invoking a function

```
mean(1:10, na.rm = TRUE)
```

```
[1] 5.5
```

```
mean(, TRUE, x = 1:10)
```

```
[1] 5.5
```

```
args <- list(1:10, na.rm = TRUE)
do.call(mean, args)
```

```
[1] 5.5
```

Function composition

```
square <- function(x) { x^2 }
deviation <- function(x) { x - mean(x) }
x <- runif(100)
```

Nesting:

```
sqrt(mean(square(deviation(x))))
```

```
[1] 0.3
```

Intermediate variables:

```
out <- deviation(x)
out <- square(out)
out <- mean(out)
out <- sqrt(out)
out
```

```
[1] 0.3
```

Pipe:

```
x |>
  deviation() |>
  square() |>
  mean() |>
  sqrt()
```

```
[1] 0.3
```

Lexical scoping

Names defined inside a function mask names defined outside a function.

```
x <- 10
y <- 20
g02 <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
g02()
```

```
[1] 1 2
```

Lexical scoping

Names defined inside a function mask names defined outside a function.

```
x <- 2
g03 <- function() {
  y <- 1
  c(x, y)
}
g03()
```

```
[1] 2 1
```

```
# And this doesn't change the previous value of y
y
```

```
[1] 20
```

Lexical scoping

Names defined inside a function mask names defined outside a function.

```
x <- 1
g04 <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
g04()
```

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

Functions versus variables

```
g07 <- function(x) { x + 1 }
g08 <- function() {
  g07 <- function(x) { x + 100 }
  g07(10)
}
g08()
```

```
[1] 110
```

```
g09 <- function(x) { x + 100 }
g10 <- function() {
  g09 <- 10
  g09(g09)
}
g10()
```

```
[1] 110
```

A fresh start

What happens to values between invocations of a function?

```
g11 <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  a
}

g11()
```

```
[1] 1
```

```
g11()
```

```
[1] 1
```

Dynamic lookup

```
g12 <- function() { x + 1 }
x <- 15
g12()
```

```
[1] 16
```

```
x <- 20
g12()
```

```
[1] 21
```

```
codetools::findGlobals(g12)
```

```
[1] "{" "+" "x"
```

Dynamic lookup

```
g12 <- function() { x + 1 }
x <- 15
g12()
```

```
[1] 16
```

```
x <- 20
g12()
```

```
[1] 21
```

```
codetools::findGlobals(g12)
```

```
[1] "{" "+" "x"
```

It is good practice to pass all the inputs to a function as arguments.

Lazy evaluation

This code doesn't generate an error because `x` is never used:

```
h01 <- function(x) {
  10
}
h01(stop("This is an error!"))
```

```
[1] 10
```

Promises

Lazy evaluation is powered by a data structure called a **promise**.

A promise has three components:

- An expression, like `x + y`, which gives rise to the delayed computation.
- An environment where the expression should be evaluated
- A value, which is computed and cached the first time a promise is accessed when the expression is evaluated in the specified environment.

Promises

```
y <- 10
h02 <- function(x) {
  y <- 100
  x + 1
}
h02(y)
```

```
[1] 11
```

Promises

```
double <- function(x) {
  message("Calculating...")
  x * 2
}
h03 <- function(x) {
  c(x, x)
}
h03(double(20))
```

```
Calculating...
```

```
[1] 40 40
```

Promises

```
double <- function(x) {
  message("Calculating...")
  x * 2
}
h03 <- function(x) {
  c(x, x)
}
h03(double(20))
```

```
Calculating...
```

```
[1] 40 40
```

Promises are like a quantum state: any attempt to inspect them with R code will force an immediate evaluation, making the promise disappear.

Default arguments

Thanks to lazy evaluation, default values can be defined in terms of other arguments, or even in terms of variables defined later in the function:

```
h04 <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100
  c(x, y, z)
}
h04()
```

```
[1]   1   2 110
```

Default arguments

Thanks to lazy evaluation, default values can be defined in terms of other arguments, or even in terms of variables defined later in the function:

```
h04 <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100
  c(x, y, z)
}
h04()
```

```
[1]   1   2 110
```

Not recommended!

Exercises

14 In `hist()`, the default value of `xlim` is `range(breaks)`, the default value for `breaks` is `"Sturges"`, and

```
range("Sturges")
```

```
[1] "Sturges" "Sturges"
```

Explain how `hist()` works to get a correct `xlim` value.

Exercises

15 Explain why this function works. Why is it confusing?

```
show_time <- function(x = stop("Error!")) {
  stop <- function(...) Sys.time()
  print(x)
}
show_time()
```

```
[1] "2024-05-21 11:35:40 UTC"
```

. . . (dot-dot-dot)

Allows for any number of additional arguments.

You can use . . . to pass additional arguments to another function.

```
i01 <- function(y, z) {
  list(y = y, z = z)
}
i02 <- function(x, ...) {
  i01(...)
}
str(i02(x = 1, y = 2, z = 3))
```

```
List of 2
 $ y: num 2
 $ z: num 3
```

. . . (dot-dot-dot)

`list(...)` evaluates the arguments and stores them in a list:

```
i04 <- function(...) {
  list(...)
}
str(i04(a = 1, b = 2))
```

```
List of 2
 $ a: num 1
 $ b: num 2
```

. . . (dot-dot-dot)

- Allows you to pass arguments to a function called within your function, without having to list them all explicitly.

. . . (dot-dot-dot)

- Allows you to pass arguments to a function called within your function, without having to list them all explicitly.

Two downsides:

- When you use it to pass arguments to another function, you have to carefully explain to the user where those arguments go.
- A misspelled argument will not raise an error. This makes it easy for typos to go unnoticed:

```
sum(1, 2, NA, na_rm = TRUE)
```

```
[1] NA
```

Exercises

16 Explain the following results:

```
sum(1, 2, 3)
```

```
[1] 6
```

```
mean(1, 2, 3)
```

```
[1] 1
```

```
sum(1, 2, 3, na.omit = TRUE)
```

```
[1] 7
```

```
mean(1, 2, 3, na.omit = TRUE)
```

```
[1] 1
```

Exiting a function

Most functions exit in one of two ways:

- return a value, indicating success
- throw an error, indicating failure.

Implicit versus explicit returns

Implicit return, where the last evaluated expression is the return value:

```
j01 <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
j01(5)
```

```
[1] 0
```

```
j01(15)
```

```
[1] 10
```

Implicit versus explicit returns

Explicit return, by calling `return()`:

```
j02 <- function(x) {
  if (x < 10) {
    return(0)
  } else {
    return(10)
  }
}
j02(5)
```

```
[1] 0
```

```
j02(15)
```

```
[1] 10
```

Invisible values

Most functions return visibly: calling the function in an interactive context prints the result.

```
j03 <- function() { 1 }
j03()
```

```
[1] 1
```

However, you can prevent automatic printing by applying `invisible()` to the last value:

```
j04 <- function() { invisible(1) }
j04()
```

Invisible values

The most common function that returns invisibly is `<-`:

```
a <- 2
(a <- 2)
```

```
[1] 2
```

This is what makes it possible to chain assignments:

```
a <- b <- c <- d <- 2
```

In general, any function called primarily for a side effect (like `<-`, `print()`, or `plot()`) should return an invisible value (typically the value of the first argument).

Errors

If a function cannot complete its assigned task, it should throw an error with `stop()`, which immediately terminates the execution of the function.

```
j05 <- function() {
  stop("I'm an error")
  return(10)
}
j05()
```

```
Error in j05(): I'm an error
```

Exit handlers

```r
j06 <- function(x) {
  cat("Hello\n")
  on.exit(cat("Goodbye!\n"), add = TRUE)
  if (x) {
    return(10)
  } else {
    stop("Error")
  }
}
```

```r
j06(TRUE)
```

```
Hello
Goodbye!
[1] 10
```

```r
j06(FALSE)
```

```
Hello
Error in j06(FALSE): Error
Goodbye!
```

Exit handlers

`on.exit()` allows you to add clean-up code

```
with_dir <- function(dir, code) {
  old <- setwd(dir)
  on.exit(setwd(old), add = TRUE)
  code
}
getwd()
```

```
[1] "/home/runner/work/arp/arp/week2"
```

```
with_dir("~", getwd())
```

```
[1] "/home/runner"
```

```
getwd()
```

```
[1] "/home/runner/work/arp/arp/week2"
```

Function forms

To understand computations in R, two slogans are helpful:

- *Everything that exists is an object.*
- *Everything that happens is a function call.*

— John Chambers

Function forms

- **prefix**: the function name comes before its arguments, like `foofy(a, b, c)`.
- **infix**: the function name comes in between its arguments, like `x + y`.
- **replacement**: functions that replace values by assignment, like `names(df) <- c("a", "b", "c")`.
- **special**: functions like `[[`, `if`, and `for`.

Rewriting to prefix form

Everything can be written in prefix form.

```
x + y
`+`(x, y)

names(df) <- c("x", "y", "z")
`names<-`(df, c("x", "y", "z"))

for(i in 1:10) print(i)
`for`(i, 1:10, print(i))
```

Don’t be evil!

```
`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
```

```
 [1] 3 3 3 3 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 3 3 3
[36] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

Prefix form

You can specify arguments in three ways:

- By position, like `help(mean)`.
- By name, like `help(topic = mean)`.
- Using partial matching, like `help(top = mean)`.

Exercises

17 Clarify the following list of odd function calls:

```
x <- sample(replace = TRUE, 20, x = c(1:10, NA))
y <- runif(min = 0, max = 1, 20)
cor(m = "k", y = y, u = "p", x = x)
```

Infix functions

Functions with 2 arguments, and the function name comes between the arguments:

`:`, `::`, `:::`, `$`, `@`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&`, `&&`, `|`, `||`, `~`, `<-`, and `<<-`.

```
1 + 2
```

```
[1] 3
```

```
`+`(1, 2)
```

```
[1] 3
```

Infix functions

You can also create your own infix functions that start and end with %.

```
`%+%` <- function(a, b) paste0(a, b)
"new " %+% "string"
```

```
[1] "new string"
```

Replacement functions

- Replacement functions act like they modify their arguments in place, and have the special name `xxx<-`.
- They must have arguments named `x` and `value`, and must return the modified object.

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
x
```

```
 [1]  1  5  3  4  5  6  7  8  9 10
```

Replacement functions

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- 10
x
```

```
 [1] 10  5  3  4  5  6  7  8  9 10
```

When you write `modify(x, 1) <- 10`, behind the scenes R turns it into:

```
x <- `modify<-`(x, 1, 10)
```

Outline

Environment basics

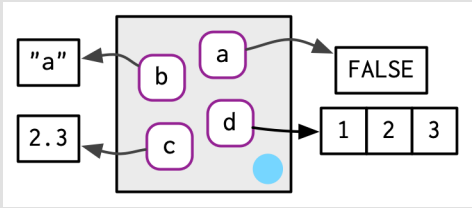
Environments are data structures that power scoping.

An environment is similar to a named list, with four important exceptions:

- Every name must be unique.
- The names in an environment are not ordered.
- An environment has a parent.
- Environments are not copied when modified.

Environment basics

```
library(rlang)
e1 <- env(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3,
)
```



Special environments

- The **current environment** is the environment in which code is currently executing.
- The **global environment** is where all interactive computation takes place. Your “workspace”.

Parent environments

- Every environment has a parent.
- If a name is not found in an environment, R looks in the parent environment.
- The ancestors of the global environment include every attached package.

```
env_parents(e1, last = empty_env())
```

```
 [[1]] $ <env: global>
 [[2]] $ <env: package:rlang>
 [[3]] $ <env: package:dplyr>
 [[4]] $ <env: package:stats>
 [[5]] $ <env: package:graphics>
 [[6]] $ <env: package:grDevices>
 [[7]] $ <env: package:datasets>
 [[8]] $ <env: renv:shims>
 [[9]] $ <env: package:utils>
[[10]] $ <env: package:methods>
[[11]] $ <env: Autoloads>
[[12]] $ <env: package:base>
[[13]] $ <env: empty>
```

Super assignment

- Regular assignment (`<-`) creates a variable in the current environment.
- Super assignment (`<<-`) modifies a variable in a parent environment.
- If it can't find an existing variable, it creates one in the global environment.

Package environments

- Every package attached becomes one of the parents of the global environment (in order of attachment).

```
search()
```

```
 [1] ".GlobalEnv"        "package:rlang"     "package:dplyr"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:datasets"  "renv:shims"        "package:utils"
[10] "package:methods"   "Autoloads"         "package:base"
```

- Attaching a package changes the parent of the global environment.
- `Autoloads` uses delayed bindings to save memory by only loading package objects when needed.

Function environments

A function binds the current environment when it is created.

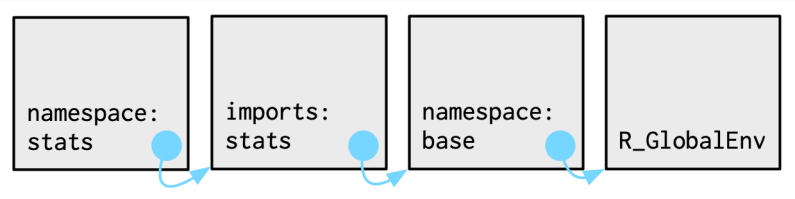
```
y <- 1
f <- function(x) {
  env_print(current_env())
  x + y
}
f(2)
```

```
<environment: 0x555a55907808>
Parent: <environment: global>
Bindings:
* x: <lazy>

[1] 3
```

Namespaces

- Package environment: how an R user finds a function in an attached package (only includes exports)
- Namespace environment: how a package finds its own objects (includes non-exports as well)
- Each namespace environment has an imports environment (controlled via NAMESPACE file).



Caller environments

```
f <- function(x) {
  g(x = 2)
}
g <- function(x) {
  h(x = 3)
}
h <- function(x) {
  stop()
}
```

```
f(x = 1)
#> Error: in h(x = 3)
traceback()
#> 4: stop() at #3
#> 3: h(x = 3) at #3
#> 2: g(x = 2) at #3
#> 1: f(x = 1)
```

Lazy evaluation

```
a <- function(x) b(x)
b <- function(x) c(x)
c <- function(x) x
a(f())
#> Error: in h(x = 3)
traceback()
#> 7: stop() at #3
#> 6: h(x = 3) at #3
#> 5: g(x = 2) at #3
#> 4: f() at #1
#> 3: c(x) at #1
#> 2: b(x) at #1
#> 1: a(f())
unused argument (clas
```