ETC4500/ETC5450
Advanced R programming

Week 1: Foundations of R programming

Outline

Introduction

Expectations

- You know R and RStudio
- You have a basic understanding of programming (for loops, if statements, functions)
- You can use Git and GitHub (https://happygitwithr.com)

Unit resources

- Everything on **https://arp.numbat.space**
- Assignments submitted on Github Classroom
- Discussion on Ed

GitHub

- Use your monash edu address.
- Apply to GitHub Education as a student (https://github.com/education/students).
- Gives you free access to private repos and GitHub Copilot.
- Add GitHub Copilot to RStudio settings, or sign into GitHub in Positron.

R history

- S (1976, Chambers, Becker and Wilks; Bell Labs, USA)
- S-PLUS (1988, Doug Martin; Uni of Washington, USA)
- R (1993, Ihaka and Gentleman; Uni of Auckland, NZ)

R history

- S (1976, Chambers, Becker and Wilks; Bell Labs, USA)
- S-PLUS (1988, Doug Martin; Uni of Washington, USA)
- R (1993, Ihaka and Gentleman; Uni of Auckland, NZ)

R influenced by

- Lisp (functional programming, environments, dynamic typing)
- Scheme (functional programming, lexical scoping)
- S and S-PLUS (syntax)

Why R?

- Free, open source, and on every major platform.
- A diverse and welcoming community
- A massive set of packages, often cutting-edge.
- Powerful communication tools (Shiny, Rmarkdown, quarto)
- RStudio and Positron IDEs
- Deep-seated language support for data analysis.
- A strong foundation of functional programming.
- Posit
- Easy connection to high-performance programming languages like C, Fortran, and C++.

R challenges

- R users are not usually programmers. Most R code by ordinary users is not very elegant, fast, or easy to understand.
- R users more focused on results than good software practices.
- R packages are inconsistent in design.
- R can be slow.

Outline

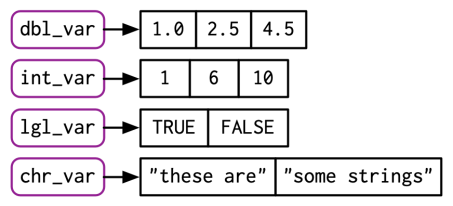
Scalars

- **Logicals**: `TRUE` or `FALSE`, or abbreviated (`T` or `F`).
- **Doubles**: decimal (`0.1234`), scientific (`1.23e4`), or hexadecimal (`0xcafe`). Special values: `Inf`, `-Inf`, and `NaN` (not a number).
- **Integers**: `1234L`, `1e4L`, or `0xcafeL`. Can not contain fractional values.
- **Strings**: `"hi"` or `'bye'`. Special characters are escaped with `\`.

Making longer vectors with `c()`

Use `c()` to create vectors.

```r
lgl_var <- c(TRUE, FALSE)
int_var <- c(1L, 6L, 10L)
dbl_var <- c(1, 2.5, 4.5)
chr_var <- c("these are", "some strings")
```



When the inputs are atomic vectors, `c()` always flattens.

```r
c(c(1, 2), c(3, 4))
```

```
[1] 1 2 3 4
```

Atomic vectors

- Four primary types of atomic vectors: `logical`, `double`, `integer`, and `character` (which contains strings).
- Two rare types: `complex`, `raw`.
- Collectively `integer` and `double` vectors are known as `numeric` vectors
- `NULL` is like a zero length vector
- Scalars are just vectors of length 1
- Every vector can also have **attributes**: a named list of arbitrary metadata.
- The **dimension** attribute turns vectors into matrices and arrays.

Types and length

You can determine the type of a vector with `typeof()` and its length with `length()`.

```
typeof(lgl_var)
```

```
[1] "logical"
```

```
typeof(int_var)
```

```
[1] "integer"
```

```
typeof(dbl_var)
```

```
[1] "double"
```

```
typeof(chr_var)
```

```
[1] "character"
```

Missing values

Most computations involving a missing value will return another missing value.

```
NA > 5
```

```
[1] NA
```

```
10 * NA
```

```
[1] NA
```

```
!NA
```

```
[1] NA
```

Missing values

Exceptions:

```r
NA ^ 0
```

```
[1] 1
```

```r
NA | TRUE
```

```
[1] TRUE
```

```r
NA & FALSE
```

```
[1] FALSE
```

Missing values

Use `is.na()` to check for missingness

```
x <- c(NA, 5, NA, 10)
x == NA
```

```
[1] NA NA NA NA
```

```
is.na(x)
```

```
[1]  TRUE FALSE  TRUE FALSE
```

There are actually four missing values: `NA` (logical), `NA_integer_` (integer), `NA_real_` (double), and `NA_character_` (character).

Coercion

- For atomic vectors, all elements must be the same type.
- When you combine different types they are **coerced** in a fixed order: logical $\rightarrow$ integer $\rightarrow$ double $\rightarrow$ character.

```
str(c("a", 1))
```

```
 chr [1:2] "a" "1"
```

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
```

```
[1] 0 0 1
```

```
sum(x)
```

```
[1] 1
```

```
as.integer(c("1", "1.5", "a"))
```

```
[1]  1  1 NA
```

Exercises

1. Predict the output of the following:

```
c(1, FALSE)
c("a", 1)
c(TRUE, 1L)
```

2. Why is `1 == "1"` true? Why is `-1 < FALSE` true? Why is `"one" < 2` false?

3. Why is the default missing value, `NA`, a logical vector? What's special about logical vectors? (Hint: think about `c(FALSE, NA_character_)`.)

Getting and setting attributes

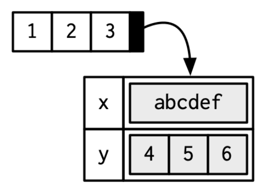
- You can think of attributes as name-value pairs that attach metadata to an object.
- Individual attributes can be retrieved and modified with `attr()`, or retrieved en masse with `attributes()`, and set en masse with `structure()`.

```
a <- 1:3
attr(a, "x") <- "abcdef"
a
```

```
[1] 1 2 3
attr(,"x")
[1] "abcdef"
```

Getting and setting attributes

```r
attr(a, "y") <- 4:6
str(attributes(a))
```




```
List of 2
 $ x: chr "abcdef"
 $ y: int [1:3] 4 5 6
```

```r
# Or equivalently
a <- structure(
  1:3,
  x = "abcdef",
  y = 4:6
)
str(attributes(a))
```

```
List of 2
 $ x: chr "abcdef"
 $ y: int [1:3] 4 5 6
```

Names

- Names are a type of attribute.
- You can name a vector in three ways:

```
# When creating it:
x <- c(a = 1, b = 2, c = 3)

# By assigning a character vector to names()
x <- 1:3
names(x) <- c("a", "b", "c")

# Inline, with setNames():
x <- setNames(1:3, c("a", "b", "c"))
```

```
x
```

```
a b c
1 2 3
```

Names

- Avoid using `attr(x, "names")` as it requires more typing and is less readable than `names(x)`.
- You can remove names from a vector by using `x <- unname(x)` or `names(x) <- NULL`.

Dimensions

- Adding a `dim` attribute to a vector allows it to behave like a 2-dimensional **matrix** or a multi-dimensional **array**.
- You can create matrices and arrays with `matrix()` and `array()`, or by using the assignment form of `dim()`:

```
# Two scalar arguments specify row and column sizes
x <- matrix(1:6, nrow = 2, ncol = 3)
x
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Dimensions

```r
# One vector argument to describe all dimensions
y <- array(1:12, c(2, 3, 2))
y
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Dimensions

```r
# You can also modify an object in place by setting dim()
z <- 1:6
dim(z) <- c(3, 2)
z
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

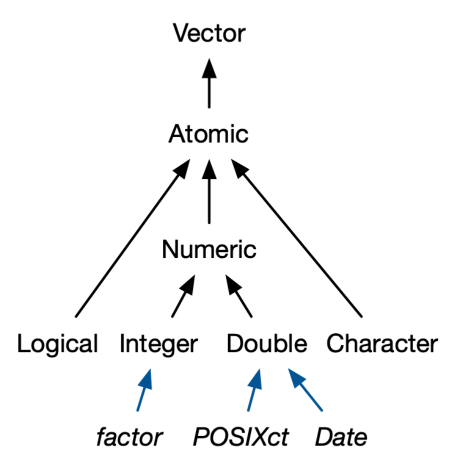
Exercises

4. What does `dim()` return when applied to a 1-dimensional vector?
5. When might you use `NROW()` or `NCOL()`?
6. How would you describe the following three objects? What makes them different from `1:5`?

```r
x1 <- array(1:5, c(1, 1, 5))
x2 <- array(1:5, c(1, 5, 1))
x3 <- array(1:5, c(5, 1, 1))
```

S3 atomic vectors

- `class` is a vector attribute.
- It turns object into **S3 object**.
- Four important S3 vectors:
  - **factor** vectors.
  - **Date** vectors with day resolution.
  - **POSIXct** vectors for date-times.
  - **difftime** vectors for durations.



Factors

- A vector that can contain only predefined values.
- Used to store categorical data.
- Built on top of an integer vector with two attributes: a `class`, "factor", and `levels`, which defines the set of allowed values.

```
x <- factor(c("a", "b", "b", "a"))
x
```

```
[1] a b b a
Levels: a b
```

Factors

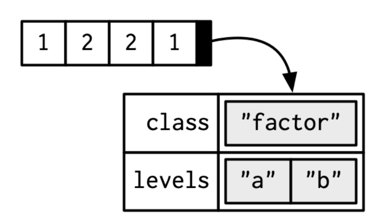
```
typeof(x)
```

```
[1] "integer"
```

```
attributes(x)
```

```
$levels
[1] "a" "b"

$class
[1] "factor"
```



Factors

```r
sex_char <- c("m", "m", "m")
sex_factor <- factor(sex_char, levels = c("m", "f"))

table(sex_char)
```

```
sex_char
m
3
```

```r
table(sex_factor)
```

```
sex_factor
m f
3 0
```

Factors

- Be careful: some functions convert factors to integers!
- ggplot preserves ordering of levels in graphs – useful to reorder panels or legends.
- Ordered factors are useful when the order of levels is meaningful.

```r
grade <- ordered(c("b", "b", "a", "c"), levels = c("c", "b", "a"))
grade
```

```
[1] b b a c
Levels: c < b < a
```

Dates

- Date vectors are built on top of double vectors.
- Class “Date” with no other attributes:

```r
today <- Sys.Date()

typeof(today)
```

```
[1] "double"
```

```r
attributes(today)
```

```
$class
[1] "Date"
```

Dates

The value of the double (which can be seen by stripping the class), represents the number of days since 1970-01-01 (the "Unix Epoch").

```r
date <- as.Date("1970-02-01")
unclass(date)
```

```
[1] 31
```

Date-times

- Base R provides two ways of storing date-time information, POSIXct, and POSIXlt.
- “POSIX” is short for Portable Operating System Interface
- “ct” stands for calendar time; “lt” for local time
- POSIXct vectors are built on top of double vectors, where the value represents the number of seconds since 1970-01-01.

```r
now_ct <- as.POSIXct("2018-08-01 22:00", tz = "UTC")
now_ct
```

```
[1] "2018-08-01 22:00:00 UTC"
```

```r
typeof(now_ct)
```

Date-times

The `tzone` attribute controls only how the date-time is formatted; it does not control the instant of time represented by the vector. Note that the time is not printed if it is midnight.

```
structure(now_ct, tzone = "Asia/Tokyo")
```

```
[1] "2018-08-02 07:00:00 JST"
```

```
structure(now_ct, tzone = "America/New_York")
```

```
[1] "2018-08-01 18:00:00 EDT"
```

```
structure(now_ct, tzone = "Australia/Lord_Howe")
```

```
[1] "2018-08-02 08:30:00 +1030"
```

Exercises

7. What sort of object does `table()` return? What is its type? What attributes does it have? How does the dimensionality change as you tabulate more variables?

8. What happens to a factor when you modify its levels?

```
f1 <- factor(letters)
levels(f1) <- rev(levels(f1))
```

9. What does this code do? How do `f2` and `f3` differ from `f1`?

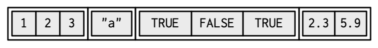
```
f2 <- rev(factor(letters))
f3 <- factor(letters, levels = rev(letters))
```

Outline

Lists

- More complex than atomic vectors
- Elements are *references* to objects of any type

```r
l1 <- list(
  1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9)
)
typeof(l1)
```




```
[1] "list"
```

```r
str(l1)
```

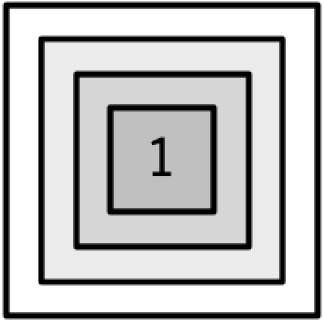
```
List of 4
 $ : int [1:3] 1 2 3
 $ : chr "a"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : num [1:2] 2.3 5.9
```

Lists

- Lists can be recursive: a list can contain other lists.

```
l3 <- list(list(list(1)))
str(l3)
```

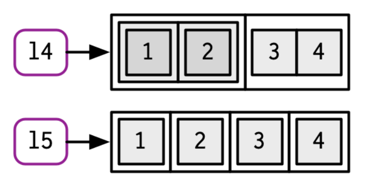
```
List of 1
 $ :List of 1
  ..$ :List of 1
  .. ..$ : num 1
```



Lists

- `c()` will combine several lists into one.

```
l4 <- list(list(1, 2), c(3, 4))
l5 <- c(list(1, 2), c(3, 4))
str(l4)
```




```
List of 2
 $ :List of 2
  ..$ : num 1
  ..$ : num 2
 $ : num [1:2] 3 4
```

```
str(l5)
```

```
List of 4
 $ : num 1
 $ : num 2
 $ : num 3
 $ : num 4
```

Testing and coercion

```r
list(1:3)
```

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

```r
as.list(1:3)
```

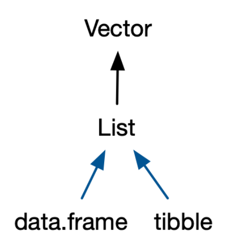
```
[[1]]
[1] 1

[[2]]
[1] 2

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

- You can turn a list into an atomic vector with `unlist()`.

Data frames and tibbles



- Most important S3 vectors built on lists: data frames and tibbles.

```
df1 <- data.frame(x = 1:3, y = letters[1:3])
typeof(df1)
```

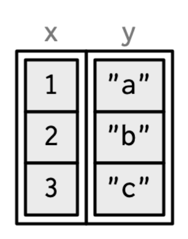
```
[1] "list"
```

```
attributes(df1)
```

```
$names
[1] "x" "y"

$class
[1] "data.frame"

$row.names
[1] 1 2 3
```



Data frames and tibbles

- A data frame has a constraint: the length of each of its vectors must be the same.
- A data frame has `rownames()` and `colnames()`. The `names()` of a data frame are the column names.
- A data frame has `nrow()` rows and `ncol()` columns. The `length()` of a data frame gives the number of columns.

Tibbles

- Modern reimagining of the data frame.
- tibbles are "lazy and surly": they do less and complain more.

```
library(tibble)
df2 <- tibble(x = 1:3, y = letters[1:3])
typeof(df2)
```

```
[1] "list"
```

```
attributes(df2)
```

```
$class
[1] "tbl_df"     "tbl"        "data.frame"

$row.names
[1] 1 2 3

$names
[1] "x" "y"
```

Creating data frames and tibbles

```r
names(data.frame(`1` = 1))
```

```
[1] "X1"
```

```r
names(tibble(`1` = 1))
```

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

Creating data frames and tibbles

```r
data.frame(x = 1:4, y = 1:2)
```

```
  x y
1 1 1
2 2 2
3 3 1
4 4 2
```

```r
tibble(x = 1:4, y = 1:2)
```

```
Error in `tibble()`:
! Tibble columns must have compatible sizes.
* Size 4: Existing data.
* Size 2: Column `y`.
i Only values of size one are recycled.
```

Creating data frames and tibbles

```r
tibble(
  x = 1:3,
  y = x * 2,
  z = 5
)
```

```
# A tibble: 3 x 3
      x     y     z
  <int> <dbl> <dbl>
1     1     2     5
2     2     4     5
3     3     6     5
```

Row names

Data frames allow you to label each row with a name, a character vector containing only unique values:

```r
df3 <- data.frame(
  age = c(35, 27, 18),
  hair = c("blond", "brown", "black"),
  row.names = c("Bob", "Susan", "Sam")
)
df3
```

```
      age  hair
Bob    35 blond
Susan  27 brown
Sam    18 black
```

Row names

- tibbles do not support row names
- convert row names into a regular column with either `rownames_to_column()`, or the `rownames` argument:

```
as_tibble(df3, rownames = "name")
```

```
# A tibble: 3 x 3
  name    age hair
  <chr> <dbl> <chr>
1 Bob      35 blond
2 Susan    27 brown
3 Sam      18 black
```

Printing

```
dplyr::starwars
```

```
# A tibble: 87 x 14
   name       height  mass hair_color skin_color eye_color birth_year sex
   <chr>       <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>
 1 Luke Skyw~    172    77 blond      fair       blue            19   male
 2 C-3PO         167    75 <NA>       gold       yellow         112   none
 3 R2-D2          96    32 <NA>       white, bl~ red             33   none
 4 Darth Vad~    202   136 none       white      yellow          41.9 male
 5 Leia Orga~    150    49 brown      light      brown           19   fema~
 6 Owen Lars     178   120 brown, gr~ light      blue            52   male
 7 Beru Whit~    165    75 brown      light      blue            47   fema~
 8 R5-D4          97    32 <NA>       white, red red             NA   none
 9 Biggs Dar~    183    84 black      light      brown           24   male
10 Obi-Wan K~    182    77 auburn, w~ fair       blue-gray       57   male
# i 77 more rows
# i 6 more variables: gender <chr>, homeworld <chr>, species <chr>,
#   films <list>, vehicles <list>, starships <list>
```

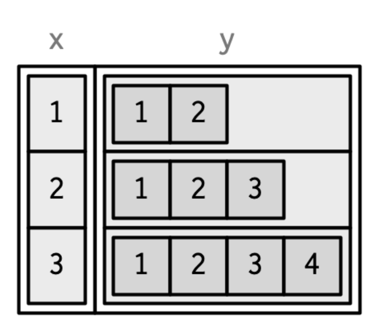
Printing

- Tibbles only show first 10 rows and all columns that fit on screen. Additional columns shown at bottom.
- Each column labelled with its type, abbreviated to 3–4 letters.
- Wide columns truncated.

List columns

```r
df <- data.frame(x = 1:3)
df$y <- list(1:2, 1:3, 1:4)
df
```

```
  x          y
1 1       1, 2
2 2    1, 2, 3
3 3 1, 2, 3, 4
```

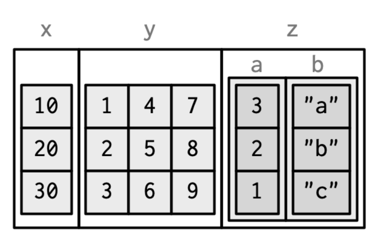



```r
tibble(
  x = 1:3,
  y = list(1:2, 1:3, 1:4)
)
```

```
# A tibble: 3 x 2
      x y
  <int> <list>
1     1 <int [2]>
2     2 <int [3]>
3     3 <int [4]>
```

Matrix and data frame columns

```r
dfm <- tibble(
  x = 1:3 * 10,
  y = matrix(1:9, nrow = 3),
  z = data.frame(a = 3:1, b = letters[1:3])
)
str(dfm)
```




```
tibble [3 x 3] (S3: tbl_df/tbl/data.frame)
 $ x: num [1:3] 10 20 30
 $ y: int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
 $ z:'data.frame':  3 obs. of  2 variables:
  ..$ a: int [1:3] 3 2 1
  ..$ b: chr [1:3] "a" "b" "c"
```

Exercises

10. What happens if you attempt to set rownames that are not unique?
11. If `df` is a data frame, what can you say about `t(df)`, and `t(t(df))`? Perform some experiments, making sure to try different column types.
12. What does `as.matrix()` do when applied to a data frame with columns of different types? How does it differ from `data.matrix()`?

NULL

```r
length(NULL)
```

```
[1] 0
```

You can test for `NULL`s with `is.null()`:

```r
x <- NULL
x == NULL
```

```
logical(0)
```

```r
is.null(x)
```

```
[1] TRUE
```

Outline

Exercises

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

Exercises

16. 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, ]
```

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

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

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

Exercises

19. How would you randomly permute the columns of a data frame?
20. 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)?
21. How could you put the columns in a data frame in alphabetical order?

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

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

```
$x


$y
```

```r
body(f02)
```

```
{
    x + y
}
```

```r
environment(f02)
```

```
<environment: R_GlobalEnv>
```

Invoking a function

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

```
[1] 5.5
```

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

```
[1] 5.5
```

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

```
[1] 5.5
```

Function composition

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

Nesting:

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

```
[1] 0.307
```

Intermediate variables:

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

```
[1] 0.307
```

Pipe:

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

```
[1] 0.307
```

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.

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

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

Functions versus variables

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

```
[1] 110
```

```r
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?

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

g11()
```

```
[1] 1
```

```r
g11()
```

```
[1] 1
```

Dynamic lookup

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

```
[1] 16
```

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

```
[1] 21
```

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

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

Dynamic lookup

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

```
[1] 16
```

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

```
[1] 21
```

```r
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

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

```
[1] 11
```

Promises

```r
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!

... (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

22 Explain the following results:

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

```
[1] 6
```

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

```
[1] 1
```

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

```
[1] 7
```

```r
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:

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

```
[1] 0
```

```r
j01(15)
```

```
[1] 10
```

Implicit versus explicit returns

Explicit return, by calling `return()`:

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

```
[1] 0
```

```r
j02(15)
```

```
[1] 10
```

Invisible values

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

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

```
[1] 1
```

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

```r
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.

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

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

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!

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

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

23 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 `<<-`.

```r
1 + 2
```

```
[1] 3
```

```r
`+`(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

```r
`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:

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

Outline

Environment basics

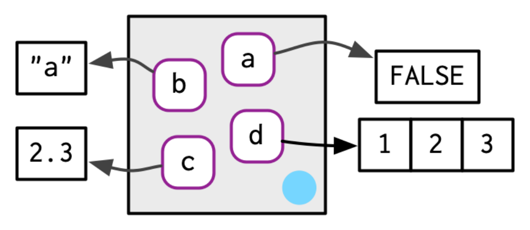
Environments are data structures that power scoping.

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:tibble>
 [[4]] $ <env: package:dplyr>
 [[5]] $ <env: package:stats>
 [[6]] $ <env: package:graphics>
 [[7]] $ <env: package:grDevices>
 [[8]] $ <env: package:datasets>
 [[9]] $ <env: renv:shims>
[[10]] $ <env: package:utils>
[[11]] $ <env: package:methods>
[[12]] $ <env: Autoloads>
[[13]] $ <env: package:base>
[[14]] $ <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).

```r
search()
```

```
 [1] ".GlobalEnv"        "package:rlang"     "package:tibble"
 [4] "package:dplyr"     "package:stats"     "package:graphics"
 [7] "package:grDevices" "package:datasets"  "renv:shims"
[10] "package:utils"     "package:methods"   "Autoloads"
[13] "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.

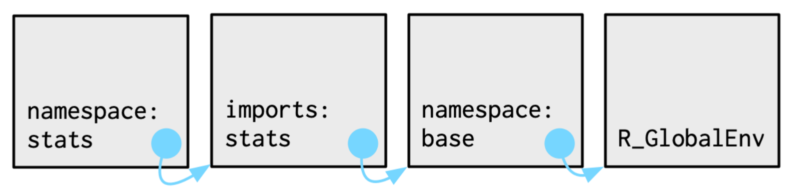
```r
y <- 1
f <- function(x) {
  env_print(current_env())
  x + y
}
f(2)
```

```
<environment: 0x5bedcba0e3c0>
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

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

```r
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
```

Outline

Conditions

```
message("This is what a message looks like")
#> This is what a message looks like

warning("This is what a warning looks like")
#> Warning: This is what a warning looks like

stop("This is what an error looks like")
#> Error in eval(expr, envir, enclos): This is what an error looks like
```

Conditions

```
message("This is what a message looks like")
#> This is what a message looks like

warning("This is what a warning looks like")
#> Warning: This is what a warning looks like

stop("This is what an error looks like")
#> Error in eval(expr, envir, enclos): This is what an error looks like
```

- Ignore messages with `suppressMessages()`.
- Ignore warnings with `suppressWarnings()`.
- Ignore errors with `try()`.

try()

- Allows execution to continue even if an error occurs.
- Returns a special object that captures the error.

```r
f1 <- function(x) {
  log(x)
  10
}
f1("x")
```

```
Error in `log()`:
! non-numeric argument to mathematical function
```

```r
f2 <- function(x) {
  try(log(x))
  10
}
f2("a")
```

```
Error in log(x) : non-numeric argument to m
[1] 10
```

Handling conditions

Allow you to specify what should happen when a condition occurs.

```
tryCatch(
  error = function(cnd) {
    # code to run when error is thrown
  },
  code_to_run_while_handlers_are_active
)
withCallingHandlers(
  warning = function(cnd) {
    # code to run when warning is signalled
  },
  message = function(cnd) {
    # code to run when message is signalled
  },
  code_to_run_while_handlers_are_active
)
```

tryCatch()

```r
f3 <- function(x) {
  tryCatch(
    error = function(cnd) NA,
    log(x)
  )
}

f3("x")
```

```
[1] NA
```

withCallingHandlers()

```
f4 <- function(x) {
  withCallingHandlers(
    warning = function(cnd) cat("How did this happen?\n"),
    log(x)
  )
}

f4(-1)
```

```
How did this happen?
```

```
[1] NaN
```

Exercise

24 Explain the results of running the following code

```
show_condition <- function(code) {
  tryCatch(
    error = function(cnd) "error",
    warning = function(cnd) "warning",
    message = function(cnd) "message",
    {
      code
      5
    }
  )
}
show_condition(stop("!"))
show_condition(10)
show_condition(warning("?!"))
```

Activity

Write a function to take a single integer input and return:

- `fizz` if the number is divisible by 5
- `buzz` if the number is divisible by 7
- `fizzbuzz` if the number is divisible by both 5 and 7
- the number otherwise

Your function should contain a `stop()` if the input is not an integer.