R Functions

Functions fundamentals

Defining your own function

```
my_fun <- function(arg1, arg2) {
  body
}
```

```
add <- function(x, y = 1) {
  x + y
}
```

Anatomy of a function

```
add <- function(x, y = 1) {
  x + y
}
```

```
> formals(add)
$x

$y
[1] 1

> body(add)
{
    x + y
}

> environment(add)
<environment: R_GlobalEnv>
```

Output: return value

```
f <- function(x) {
  if (x < 0) {
    -x
  } else {
    x
  }
}
```

```
> f(-5)
[1] 5

> f(15)
[1] 15
```

- The last expression evaluated in a function is the return value
- `return(value)` forces the function to stop execution and return `value`

Functions are objects

```
> mean2 <- mean

> mean2(1:10)
[1] 5.5

> function(x) { x + 1 }
function(x) { x + 1 }

> (function(x) { x + 1 })(2)
[1] 3
```

Summary

- Three parts of a function:
  - Arguments
  - Body
  - Environment
- Return value is the last executed expression, or the first executed `return()` statement
- Functions can be treated like usual R objects

R Functions

Let’s practice!

R Functions

Scoping in R

Scoping describes how R looks up values by name

```
> x <- 10
> x
[1] 10
```

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

```
> f()
[1] 1 2
```

If a name isn't defined inside a function, R will look one level up

```
> x <- 2
```

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

```
> g()
[1] 2 1
```

If a name isn't defined locally, or at a higher level, an error occurs

```
> rm(x)
```

Remove our definition of x

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

```
> g()
Error in g() : object 'x' not found
```

Scoping describes where, not when, to look for a value

```
> f <- function() x

> x <- 15

> f()
[1] 15

> x <- 20

> f()
[1] 20
```

Lookup works the same for functions

```
> l <- function(x) x + 1

> m <- function() {
    l <- function(x) x * 2
    l(10)
  }

> m()
[1] 20
```

```
> c <- 3

> c(c = c)
```

Each call to a function has its own clean environment

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

```
> j()
[1] 1

> j()
[1] 1

> a
Error: object 'a' not found
```

Summary

- When you call a function, a new environment is made for the function to do its work
- The new environment is populated with the argument values
- Objects are looked for first in this environment
- If they are not found, they are looked for in the environment that the function was created in

R Functions

Let’s practice!

R Functions

Data structures

Two types of vectors in R

- **Atomic vectors** of six types: logical, integer, double, character, complex, and raw
- **Lists**, a.k.a recursive vectors, because lists can contain other lists
- Atomic vectors are homogeneous, lists can be heterogeneous

Every vector has two key properties

```
# Its type, find with typeof()
> typeof(letters)
[1] "character"

> typeof(1:10)
[1] "integer"

# Its length, find with length()
> length(letters)
[1] 26

> x <- list("a", "b", 1:10)
> length(x)
[1] 3
```

Missing values

```
> typeof(NULL)
[1] "NULL"
> length(NULL)
[1] 0

> typeof(NA)
[1] "logical"
> length(NA)
[1] 1
```

- `NULL` often used to indicate the absence of a vector
- `NA` used to indicate the absence of a value in a vector, a.k.a. a missing value

NAs inside vectors

```
> x <- c(1, 2, 3, NA, 5)

> x
[1]  1  2  3 NA  5

> is.na(x)
[1] FALSE FALSE FALSE  TRUE FALSE
```

Missing values are contagious

```
> NA + 10
[1] NA

> NA / 2
[1] NA

> NA > 5
[1] NA

> 10 == NA
[1] NA

> NA == NA
[1] NA
```

Lists

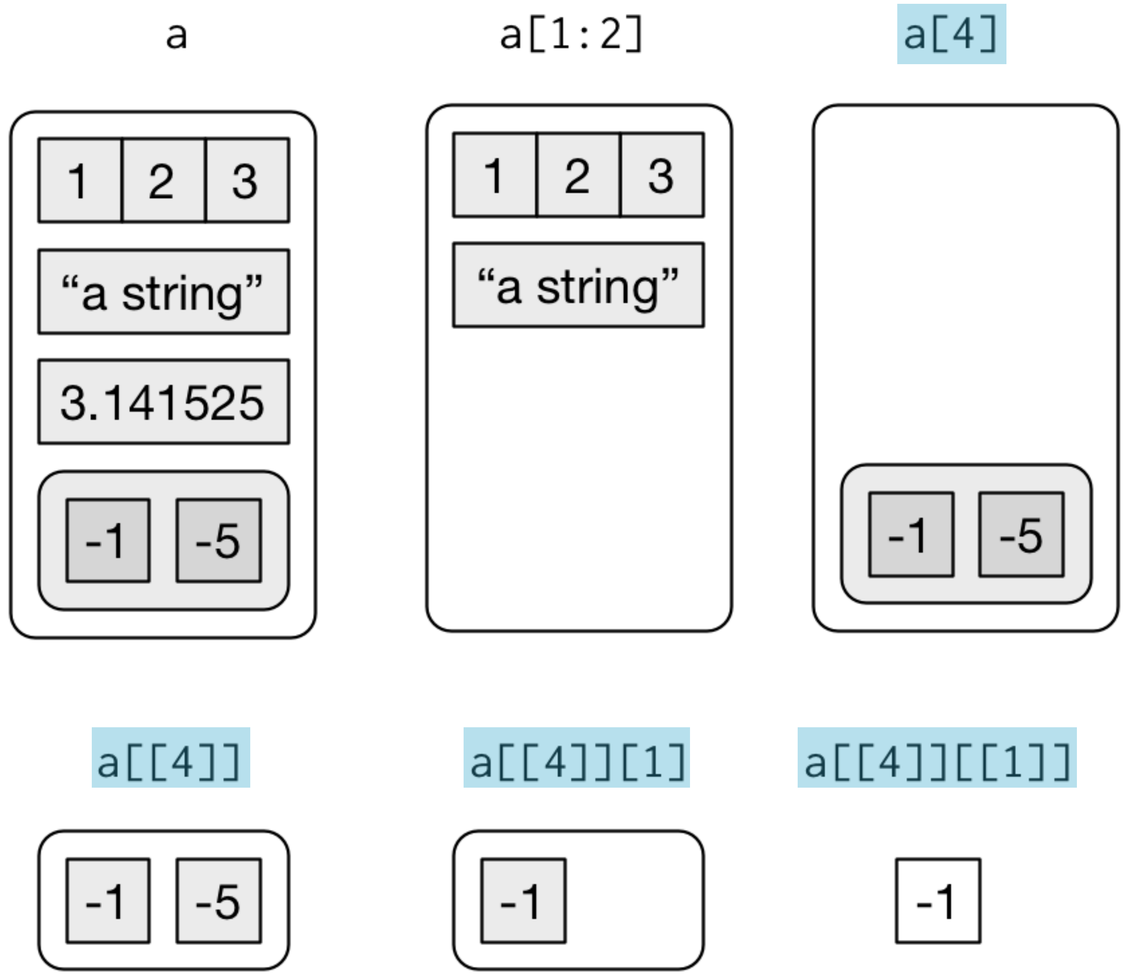
- Useful because they can contain heterogeneous objects
- Complicated return objects are often lists, i.e. from `lm()`
- Created with `list()`
- Subset with `[`, `[[` or `$`
  - `[` extracts a sublist
  - `[[` and `$` extract elements, remove a level of hierarchy

Subsetting lists

```
> a <- list(
    a = 1:3,
    b = "a string",
    c = pi,
    d = list(-1, -5)
  )

> str(a[4])
List of 1
 $ d:List of 2
  ..$ : num -1
  ..$ : num -5

> str(a[[4]])
List of 2
 $ : num -1
 $ : num -5
```



Source: Garrett Grolemund & Hadley Wickham, R for Data Science

R Functions

Let’s practice!

R Functions

for loops

for loops in R

```
> primes_list <- list(2, 3, 5, 7, 11, 13)

> for (i in 1:length(primes_list)) {
    print(primes_list[[i]])
  }
[1] 2
[1] 3
[1] 5
[1] 7
[1] 11
[1] 13
```

Parts of a for loop

```
for (i in 1:length(primes_list)) {
  print(primes[[i]])
}
```

sequence

Parts of a for loop

```
for (i in 1:length(primes_list)) {
  print(primes[[i]])
}
```

body

Parts of a for loop

```
for (i in 1:length(primes_list)) {
  print(primes[[i]])
}
```

output?

Looping over columns in a data frame

```
> df <- data.frame(
    a = rnorm(10),
    b = rnorm(10),
    c = rnorm(10),
    d = rnorm(10)
  )

> for (i in 1:ncol(df)) {
    print(median(df[[i]]))
  }
```

sequence

Looping over columns in a data frame

```
> df <- data.frame(
    a = rnorm(10),
    b = rnorm(10),
    c = rnorm(10),
    d = rnorm(10)
  )

> for (i in 1:ncol(df)) {
    print(median(df[[i]]))
  }
```

body

Looping over columns in a data frame

```
> df <- data.frame(
    a = rnorm(10),
    b = rnorm(10),
    c = rnorm(10),
    d = rnorm(10)
  )

> for (i in 1:ncol(df)) {
    print(median(df[[i]]))
  }
```

output?

Moving forward

- A safer way to generate the sequence using `seq_along()`
- Saving output instead of printing it

R Functions

Let’s practice!