Advanced R programming

(by Hadley Wickham)

http://adv-r.had.co.nz

Contents

Part I
Programming

1 Introduction

◇ Become a R programmer from a R user.

◇ There are two meta-techniques that are tremendously helpful for improving your skills as an R programmer: *reading the source*, and *adopting a scientific mindset*.

  ▷ **Reading source code** is a tremendously useful technique because it exposes you to new ways of doing things. Over time you'll develop a sense of taste as an R programmer, and even if you find something your taste violently objects to, it's still helpful: emulate the things you like and avoid the things you don't like.

  ▷ It's a great idea to start by reading the source code for the functions and packages that you use most frequently.

  ▷ If you don't understand how something works, develop a hypothesis, design some experiments, run them and record the results. This exercise is extremely useful if you can't figure it out and need to get help from others: you can easily show what you tried, and when you learn the right answer, you'll be mentally prepared to update your world view.

◇ Recommended reading

  ▷ R has aspects of both functional and object-oriented (OO) programming languages, and learning how these aspects are expressed in R will help you translate your existing knowledge from other programming languages, and to help you identify areas where you can improve.

  ▷ *The Structure and Interpretation of Computer Programs* (SICP)`http://mitpress.mit.edu/sicp/full-text/book/book.html` is particularly helpful.

  ▷ *Concepts, Techniques and Models of Computer Programming* by Peter van Roy and Sef Haridi. It helps to understand that R's copy-on-modify semantics make it substantially easier to reason about code, and while the current implementation in R is not very efficient, that it is a solvable problem.

  ▷ *The Pragmatic Programmer*, by Andrew Hunt and David Thomas. This book is program language agnostic, and provides great advice for how to be a better programmer.

2 Foundations

2.1 Data structures

- ◇ R's base data structures are summarized in the table below, organized by their dimensionality and whether they're homogeneous (all contents must be of the same type) or heterogeneous (the contents can be of different types):

	Homogeneous	**Heterogeneous**
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

  - ▷ Note that R has no scalar, or 0-dimensional, types. All scalars (single numbers or strings) are length-one vectors.
  - ▷ When trying to understand the structure of an arbitrary object in R your most important tool is str(), short for structure: it gives a compact human readable description of any R data structure.

Vectors

- ◇ The basic data structure in R is the vector, which comes in two basic flavours: **atomic vectors** and **lists**.
  1. Atomic vector: must have the same type in contents
  2. List: the contents can have different types
- ◇ Three properties of a vector:
  1. typeof(): what it is
  2. length(): how long it is
  3. attributes(): additional arbitrary metadata, like names()
- ◇ Atomic vectors and lists are the building blocks for higher dimensional data structures. Atomic vectors extend to matrices and arrays, and lists are used to create data frames
- ◇ Each type of vector comes with an as.* coercion function and an is.* testing function. But beware. For historical reasons, is.vector() returns TRUE only if the object is a vector with no attributes apart from names. Use is.atomic(x) || is.list(x) to test if an object is actually a vector.

1. **Atomic vectors**
   - ◇ **Missing values** are specified with NA, which is a logical vector of length 1. NA will always be coerced to the correct type with c(), or you can create NA's of specific types with NA_real_ (double), NA_integer_ and NA_character_.
   - ◇ **Types and tests**: typeof(), is.character(), is.double(), is.integer(), is.logical(), or, more generally, is.atomic().
   - ◇ **Coercion**: When attempt to combine different types in atomic vector they will be coerced to the lowest common type: (ordered from low to high) character, double, integer and logical.
     - ▷ You can manually force one type of vector to another using a coercion function: as.character(), as.double(), as.integer(), as.logical().

2. **Lists**

   - ◇ Lists are different from atomic vectors in that they can contain any other type of vector, including lists. You construct them using `list()` instead of `c()`.
   - ◇ Lists are sometimes called **recursive vectors**, because a list can contain other lists. This makes them fundamentally different from atomic vectors.
   - ◇ `c()` will combine several lists into one. If given a combination of atomic vectors and lists, c() will coerce the vectors to list before combining them. E.g. "`y <- c(list(1, 2), c(3, 4))`", y then is a list of 4.
   - ◇ Lists are used to build up many of the more complicated data structures in R. For example, both *data frames*, and linear models objects (as produced by lm()) are lists.
   - ◇ Using the same implicit coercion rules as for c(), you can turn a list back into an atomic vector using unlist().

3. **Factors**

   - ◇ A factor is a vector that can contain only predefined values. It is R's structure for dealing with qualitative data.
   - ◇ A factor is not an atomic vector, but it's built on top of an integer vector using an S3 class.
   - ◇ Factors have *two key attributes*: **their class()**, "factor", which controls their behavior; and **their levels()**, the set of allowed values.
   - ◇ While factors look (and often behave) like character vectors, they are actually integers under the hood and you need to be careful when treating them like strings.
   - ◇ Factors are useful when you know the possible values a variable may take, even if you don't see all values in a given dataset. Using a factor instead of a character vector makes it obvious when some groups contain no observations
   - ◇ Sometimes when reading the data into R, the numeric variable will be coerced into factor if there are some missing values encoded in special way like . or -. We can avoid this by using na.strings argument in `read.csv()`: e.g. `z <- read.csv(text="value\n12\n1\n.\n9", na.strings=".")`.
   - ◇ To coerce the vector from a factor to numeric we need first coerce it into character and then from character to numeric. E.g. `as.numeric(as.character(z$value))`
   - ◇ Unfortunately, most data loading functions in R automatically convert character vectors to factors. This is suboptimal, because there's no way for those functions to know the set of all possible levels and their optimal order. A global option (`options(stringsAsFactors = FALSE)`) is available to control this behaviour, but it's not recommended - it makes it harder to share your code, and it may have unexpected consequences when combined with other code (either from packages, or code that you're `source()`ing). [1]

- ◇ What makes `is.vector()` and `is.numeric()` fundamentally different to `is.list()` and `is.character()`?
- ◇ Why is the default (and shortest) NA a logical vector? What's special about logical vectors?

[1]Global options make code harder to understand, because they increase the number of lines you need to read to understand what a function is doing.

Attributes

- All objects can have arbitrary additional attributes. These can be thought of as a named list (with unique names). Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`. E.g. `attr(y, "my_attribute") <- "This is a vector"; str(attributes(y))`.
- The `structure()` function returns a new object with modified attributes
- By default, most attributes are lost when modifying a vector (e.g. `sum(y)`, `y[1]`); the exceptions are for the most common attributes: `names()` (character vector of element names), `class()`, `dim()` (used to turn vectors into high-dimensional structures).
- **Names**
  - You can name a vector in three ways:
    - During creation: `x <- c(a = 1, b = 2, c = 3)`
    - By modifying an existing vector: `x <- 1:3; names(x) <- c("a", "b", "c")`
    - By creating a modified vector: `x <- setNames(1:3, c("a", "b", "c"))`
  - Names should be unique, because character subsetting, the biggest reason to use names, will only return the first match.
  - Not all elements of a vector need to have a name. If any names are missing, `names()` will return an empty string for those elements. If all names are missing, `names()` will return `NULL`.
  - You can create a vector without names using `unname(x)`, or remove names in place with `names(x) <- NULL`.

Matrices and arrays

- Adding a `dim()` attribute allows an atomic vector to also be treated like a multi-dimensional array. A special case of a general array is the matrix, which has two dimensions.
- Matrices and arrays are created with `matrix()` and `array()`, or by using the replacement form of `dim()`. E.g. `array(1:12, c(2, 3, 2))`
- The basic properties `length()` and `names()` have high-dimensional generalizations that work with matrices and arrays:
  - `length()` generalizes to `nrow()` and `ncol()` for matrices, and `dim()` for arrays.
  - `names()` generalizes to `rownames()` and `colnames()` for matrices, and `dimnames()`, a list, for arrays.
- `c()` generalizes to `cbind()` and `rbind()` for matrices, and to `abind()` (provided by the `abind` package) for arrays.
- While atomic vectors are most commonly turned into matrices, the dimension attribute can also be set on lists to make list-matrices or list-arrays. `l <- list(1:3, "a", TRUE, 1.0); dim(l) <- c(2, 2)`.

Data frames

- **A data frame is a list of equal-length vectors.** This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list. This means that a data frame has `names()`, `colnames()` and `rownames()`, although `names()` and `colnames()` are the same thing. The `length()` of a data frame is the length of the underlying list and so is the same as `ncol()`, `nrow()` gives the number of rows.

- As doing subsetting, you can subset a data frame like a 1d structure (where it behaves like a list), or a 2d structure (where it behaves like a matrix).
- By default `data.frame()` converts strings into factors. Use `stringAsFactors = FALSE` to suppress this behaviour.
- `data.frame` is an S3 class, so its type reflects the underlying vector used to build it: list. (`typeof()` gives list). We should look at its `class()` or test explicitly with `is.data.frame()`.
- We can combine data frames using `cbind()` and `rbind()`. When combining column-wise, only the number of rows matters, the rownames are ignored; while when combining row-wise, the column names must match. If you want to combine data frames that may not have all the same variables, use `plyr::rbind.fill()`.
- Since a data frame is a list of vectors, it is possible for a data frame to have a column that is a list.
- Ex: `dfl <- data.frame(x = 1:3, y = I(list(1:2, 1:3, 1:4)))`, use `I()` to treat the whole list as an unit, otherwise, `data.frame()` will try to used each element in the list as a column.

2.2 Subsetting

Data types

- **Atomic vectors**: x
  - Five ways to subset x:

    1. **Positive integers**: return elements at the specified positions
       Real numbers are silently truncated to integers
    2. **Nagative integers**: omit elements at the specified positions
    3. **Logical vector**: selects elements where the corresponding logical value is TRUE.
       - If logical vector is horter than the vector being subsetted, it will be recycled to be the same length.
       - A missing value in the index always yields a missing value in the output
    4. With **nothing**: returns the origial vector unchanged.
       Not useful in 1d but very useful in 2d
    5. With **zero**: returns a zero-length vector.

  - If a vector is named, we can also subset with
    - **character vector**: retuns elements with matching names.
- **Lists**
  - Subsetting a list works in exactly the same way as subsetting an atomic vector. Subsetting a list with [will always return a list: [[and $, as described below, let you pull out the components of the list.
- **Matrices (2d) and arrays (>2d)**
  - We can subset higher-dimensional structures in three ways: with **mutiple vectors**, with **single vector**, or **with a matrix**.
  - By default, [will simplify the results to the lowest possible dimensionality.

- ▷ Because matrices and arrays are implemented as vectors with special attributes, we can also subset them with a single vector. The length of a matrix/array=nrow*ncol of it. In that case, they will behave like a vector. Arrays in R are stored in column-major order.
- ▷ Using integer matrix (or a character matrix if named), each row in the matrix specifies the location of a value, with each column corresponding to a dimension in the array being subsetted.
- ▷ `upper.tri()` returns a matrix with all uper triangle elemetns with TRUEs and others False for a matrix.

♢ **Data frames**

- ▷ Data frames possess the characteristics of both lists and matrices: if you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices.

♢ **S3 objects vs. S4 objects**

1. S3 objects are made up of *atomic vectors, arrays* and *lists*, so we can always pull apart an S3 object using the techniques described above and the knowledge you gain from `str()`.
2. There are also two additional subsetting operators that are needed for S4 objects: `@` (equivalent to `$`), and `slot()` (equivalent to `[[`). `@` is more restrictive than $ in that it will return an error if the slot does not exist.

Subsetting operators

♢ Apart from `[`, there are two other subsetting operators: `[[` and `$`. `[[` is similar to `[`, except it can only return a single value, and it allows you to pull pieces out of a list. *When combined with character subsetting, $ is a useful shorthand for [[.*

- ▷ E.g.: `b <- list(a = list(b = list(c = list(d = 1))))` `b[[c("a", "b", "c", "d")]]` is the same as `b[["a"]][["b"]][["c"]][["d"]]`
- ▷ Because data frames are lists of columns, you can use `[[` to extract a column from data frames.

♢ **Simplifying vs. preserving subsetting**

- ▷ *Simplifying subsets* return the simplest possible data structure that can represent the output.
- ▷ *Preserving subsetting* keeps the structure of the output the same as the input. It is generally better for programming because the result will always be the same type.
- ▷ Switch between subsetting and preserving for different data types

	Simplifying	**Preserving**
Vector	x[[1]]	x[1]
List	x[[1]]	x[1]
Factor	x[1:4,drop=T]	x[1:4]
Array	x[1,] or x[,1]	x[1, , drop=F] or x[, 1, drop=F]
Data frame	x[,1] or x[[1]]	x[, 1, drop=F] or x[1]

- ▷ Simplifying behaviour varies slightly between different data types:
  - ⋄ atomic vector: removes names
  - ⋄ list: return the object inside the list, not a single element list
  - ⋄ factor: drops any unused levels
  - ⋄ matrix or array: if any of the dimensions has length 1, drops that dimension.

- ◇ data frame: if output is a single column, returns a vector instead of a data frame

- ♢ **$**
  - ▷ $ is a shorthand operator, where x$y is equivalent to x[["y", exact = FALSE]].
  - ▷ One common mistake with $ is to try and use it when you have the name of a column stored in a variable: var<-"cyl"; mtcars$var # wrong; mtcars[[var]] #right
  - ▷ There's one important difference between $ and [[- $ does partial matching: x <- list(abc = 1) x$a #> [1] 1 x[["a"]] #> NULL. This can be avoided by using options(warnPartialMatchDollar = TRUE), but beware of the danger of using the global option.
- ♢ Missing/out of bounds indices

Operator	**Index**	**Atomic**	**List**
[	OOB	NA	list(NULL)
[	NA_real_	NA	list(NULL)
[	NULL	x[0]	list(NULL)
[[	OOB	Error	Error
[[	NA_real_	Error	NULL
[[	NULL	Error	Error

Subsetting and assignment

- ♢ Indexing with a blank can be useful in conjunction with assignment because it will preserve the original object class and structure.
  - ▷ mtcars[] <- lapply(mtcars, as.integer) will retrun a dataframe
  - ▷ mtcars <- lapply(mtcars, as.integer) will return a list
- ♢ With *lists*, you can use subsetting + assignment + NULL to **remove** components from a list. To **add** a literal NULL to a list, use [and list(NULL):
  - ▷ x <- list(a = 1, b = 2) x[["b"]] <- NULL, will remove b and result in a list of 1
  - ▷ y <- list(a = 1) y["b"] <- list(NULL)

Applications

- ♢ **Lookup tables (character subsetting)**
  - ▷ Character matching provides a powerful way to make lookup tables.
    x <- c("m", "f", "u", "f", "f", "m", "m")
    lookup <- c(m = "Male", f = "Female", u = NA)
    lookup[x]
    or
    c(m = "Known", f = "Known", u = "Unknown")[x]
    If you don't want names in the result, use unname() to remove them.
- ♢ **Matching and merging by hand (integer subsetting)**

- ▷ You may have a more complicated lookup table which has multiple columns of information:

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame( grade = 1:3, desc = c("Poor", "Good", "Excellent"), fail = c(T, F, F) )
# Using match
id <- match(grades, info$grade) info[id, ]
#> grade desc fail
#> 1 1 Poor TRUE
#> 2 2 Good FALSE
#> 2.1 2 Good FALSE
#> 3 3 Excellent FALSE
#> 1.1 1 Poor TRUE
# Using rownames
rownames(info) <- info$grade
info[as.character(grades), ]
#> grade desc fail
#> 1 1 Poor TRUE
#> 2 2 Good FALSE
#> 2.1 2 Good FALSE
#> 3 3 Excellent FALSE
#> 1.1 1 Poor TRUE
```

- ▷ If you have multiple columns to match on, you'll need to first collapse them to a single column (with `interaction()`, `paste()`, or `plyr::id()`). You can also use `merge()` or `plyr::join()`, which do the same thing for you.

♢ **Random samples/bootstrap (integer subsetting)**

- ▷ You can use integer indices to perform random sampling or bootstrapping of a vector or data frame. You use `sample()` to generate a vector of indices, and then use subsetting to access the values.
- ▷ The arguments of `sample()` control the number of samples to extract, and whether or not sampling with replacement is done.

♢ **Ordering (integer subsetting)**

- ▷ `order()` takes a vector as input and returns an integer vector describing how the subsetted vector should be ordered:

```
x <- c("b", "c", "a")
order(x)
#> [1] 3 1 2
x[order(x)]
#> [1] "a" "b" "c"
```

- ▷ To break ties, you can supply additional variables to `order()`, and you can change from ascending to descending order using `decreasing = TRUE`. By default, any missing values will be put at the end of the vector: however, you can remove them with `na.last = NA` or put at the front with `na.last = FALSE`.
- ▷ More concise, but less flexible, functions are available for sorting vectors, `sort()`, and data frames, `plyr::arrange()`.

- **Expanding aggregated counts (integer subsetting)**
  - Sometimes you get a data frame where identical rows have been collapsed into one and a count column has been added. rep() and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index:
    ```
    df <- data.frame(x = c(2, 4, 1), y = c(9, 11, 6), n = c(3, 5, 1))
    rep(1:nrow(df), df$n)
    #> [1] 1 1 1 2 2 2 2 2 3
    df[rep(1:nrow(df), df$n), ]
    ```
- **Removing columns from data frame (character subsetting)**
  - There are two ways to remove columns from a data frame.
    - You can set individual columns to NULL
    - Or you can subset to return only the columns you want
    - If you know the columns you don't want, use set operations to work out which colums to keep: `df[setdiff(names(df), "z")]`
- **Selecting rows based on a condition (logical subsetting)**
  - Because it allows you to easily combine conditions from multiple columns, logical subsetting is probably the mostly commonly used technique for extracting rows out of a data frame.
  - Remember to use the vector boolean operators & and |, not the short-circuiting scalar operators && and || which are more useful inside if statements. Don't forget [De Morgan's laws]:
    - !(X & Y) is the same as !X | !Y
    - !(X | Y) is the same as !X & !Y
    - !(X & !(Y | Z)) simplifies to !X | !!(Y|Z), and then to !X | Y | Z.
  - `subset()` is a specialised shorthand function for subsetting data frames, and saves some typing because you don't need to repeat the name of the data frame.
- **Boolean algebra vs sets (logical & integer subsetting)**
  - It's useful to be aware of the natural equivalence between set operations (integer subsetting) and boolean algebra (logical subsetting).
  - Using set operations is more effective when:
    - You want to find the first (or last) `TRUE`
    - You have very few `TRUE`s and very many `FALSE`s; a set representation may be faster and require less storage
  - `which()` allows you to convert a boolean representation to an integer representation.
    ```
    x <- sample(10) < 4
    which(x)
    #> [1] 3 5 6
    ```
  - `x & y <-> intersect(x, y)`
  - `x | y <-> union(x, y)`
  - `x & !y <-> setdiff(x, y)`
  - `xor(x, y) <-> setdiff(union(x,y), intersect(x, y))`

  - ▷ Also beware that x[-which(y)] is not equivalent to x[!y]: if y is all FALSE, which(y) will be integer(0) and -integer(0) is still integer(0), so you'll get no values, instead of all values. In general, avoid switching from logical to integer subsetting unless you want, for example, the first or last TRUE value.

- ♢ How would you random permute the columns of a data frame? (This is an important technique in random forests). Can you simultaneously permute the rows and columns in one step?
- ♢ 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)?

2.3 Vocabulary

The basics

- ♢ The first function to learn: ?
- ♢ Important operators and assignment: %in%, match, =, <-, <<-, $, [, [[, head, tail, subset, with, assign, get
- ♢ Comparision: all.euqal, identical, !=, ==, >, >=, <, <=, is.na, complete.cases, is.finite
- ♢ Basic math:
  - ▷ *, +, -, /, ^, %%, %/%,
  - ▷ abs, sign,
  - ▷ acos, asin, atan, atan2,
  - ▷ sin, cos, tan,
  - ▷ ceiling, floor, round, trunc, signif,
  - ▷ exp, log, log10, log2, sqrt
  - ▷ max, min, prod, sum
  - ▷ cummax, cummin, cumprod, cumsum, diff
  - ▷ pmax, pmin
  - ▷ range
  - ▷ mean, meadian, cor, sd, var
  - ▷ rle
- ♢ Functions: function, missing, on.exit, return, invisible
- ♢ Logical & sets:
  - ▷ &, |, !, xor
  - ▷ all, any
  - ▷ intersect, union, setdiff, setequal
  - ▷ which
- ♢ Vector and matrices
  - ▷ c, matrix

- ◊ Automatic coercision rules character > numeric > logical
  - ▷ length, dim, ncol, nrow
  - ▷ cbind, rbind
  - ▷ names, colnames, rownames
  - ▷ t
  - ▷ diag
  - ▷ sweep
  - ▷ as.matrix, data.matrix
- ◊ Making vectors
  - ▷ c
  - ▷ rep, rep_len
  - ▷ seq, seq_len, seq_along
  - ▷ rev
  - ▷ sample
  - ▷ choose, factorial, combn
  - ▷ (is/as).(character/numeric/logical/...)
- ◊ Lists & data.frames
  - ▷ list, unlist
  - ▷ data.frame, as.data.frame
  - ▷ split
  - ▷ expand.grid
- ◊ Control flow
  - ▷ if, &&, || (short circuiting)
  - ▷ for, while
  - ▷ next, break
  - ▷ switch
  - ▷ ifelse

Common data structures

- ◊ Date time
  - ▷ ISOdate, ISOdatetime, strftime, strptime, date
  - ▷ difftime
  - ▷ julian, months, quarters, weekdays
  - ▷ library(lubridate)
- ◊ Character manipulation

  - ▷ grep, agrep
  - ▷ gsub
  - ▷ strsplit
  - ▷ chartr
  - ▷ nchar
  - ▷ tolower, toupper
  - ▷ substr
  - ▷ paste
  - ▷ library(stringr)

- ♢ Factors
  - ▷ factor, levels, nlevels
  - ▷ reorder, relevel
  - ▷ cut, findInterval
  - ▷ interaction
  - ▷ options(stringsAsFactors = FALSE)

- ♢ Array manipulation
  - ▷ array
  - ▷ dim
  - ▷ dimnames
  - ▷ aperm l
  - ▷ ibrary(abind)

Statistics

- ♢ Ordering and tabulating
  - ▷ duplicated, unique
  - ▷ merge
  - ▷ order, rank, quantile
  - ▷ sort table, ftable

- ♢ Linear models
  - ▷ fitted, predict, resid, rstandard
  - ▷ lm, glm
  - ▷ hat, influence.measures
  - ▷ logLik, df, deviance
  - ▷ formula, ~, I
  - ▷ anova, coef, confint, vcov
  - ▷ contrasts

◇ Miscellaneous tests

- ▷ apropos("\\.test$")

◇ Random variables

- ▷ (q, p, d, r) * (beta, binom, cauchy, chisq, exp, f, gamma, geom, hyper, lnorm, logis, multinom, nbinom, norm, pois, signrank, t, unif, weibull, wilcox, birthday, tukey)

◇ Matrix algebra

- ▷ crossprod, tcrossprod
- ▷ eigen, qr, svd %
- ▷ *%, %o%, outer
- ▷ rcond
- ▷ solve

Working with R

◇ Workspace

- ▷ ls, exists, rm
- ▷ getwd, setwd
- ▷ q
- ▷ source
- ▷ install.packages, library, require

◇ Help

- ▷ help, ?
- ▷ help.search
- ▷ apropos
- ▷ RSiteSearch
- ▷ citation
- ▷ demo
- ▷ example
- ▷ vignette

◇ Debugging

- ▷ traceback
- ▷ browser
- ▷ recover
- ▷ options(error =)
- ▷ stop, warning, message
- ▷ tryCatch, try

I/O

- Output
  - print, cat message, warning
  - dput
  - format sink, capture.output
- Reading and writing data
  - data
  - count.fields
  - read.csv, write.csv
  - read.delim, write.delim
  - read.fwf readLines, writeLines
  - readRDS, saveRDS
  - load, save
  - library(foreign)
- Files and directories
  - dir
  - basename, dirname, tools::file_ext
  - file.path path.expand, normalizePath
  - file.choose
  - file.copy, file.create, file.remove, file.rename, dir.create
  - file.exists, file.info
  - tempdir, tempfile
  - download.file, library(downloader)

2.4 Functions

Components of a function

- All R functions have three parts:
  - the body(), the code inside the function.
  - the formals(), the list of arguments which controls how you can cal the function.
  - the environment(), the "map" of the location of the function's variables.
- When you print a function in R, it shows you these three important components. If the environment isn't displayed, it means that the function was created in the global environment.
- The assignment forms of body(), formals(), and environment() can also be used to modify functions.
- Like all objects in R, functions can also possess any number of additional attributes(). For example, you can can set the class() and add a custom print() method.

- One attribute used by base R is "srcref", short for *source reference*, which points to the source code used to create the function. Unlike body(), this contains code comments and other formatting.
- **Primitive functions**
  - There is one exception to the rule that functions have three components. Primitive functions, like sum, call C code directly with .Primitive() and contain no R code. Therefore their formals(), body() and environment() are all NULL.
  - Primitive functions are only found in the base package, and since they operate at a low level, they can be more efficient (primitive replacement functions don't have to make copies), and can have different rules for argument matching (e.g. switch and call).
  - This, however, comes at a cost of behaving differently from all other functions in R. Hence R core generally avoids creating them unless there is no other option.
  - Q: Create a list of all primitive functions in R.

    ```
    lt<-ls("package:base", all = TRUE)
    mark<-numeric(length(lt))
    for (i in 1:length(lt)){ mark[i]<-is.null(environment(get(lt[i]))) }
    lt[mark==1]
    ```

Lexical scoping

- Scoping is the set of rules that govern how R looks up the value of a symbol.
- Understanding scoping allows you to:
  1. build tools by composing functions
  2. overrule the usual evaluation rules and do metaprogramming
- R has two types of scoping:
  1. **lexical scoping**, implemented automatically at the language level
  2. **dynamic scoping**, used in selecting functions to save typing during interactive analysis.
- Lexical scoping looks up symbol values based on how functions were nested when they were created, not how they are nested when they are called.
- There are four basic principles behind R's implementation of lexical scoping:
  1. **name masking**
     - If a name isn't defined inside a function, R will look one level up.
     - The same rules apply if a function is defined inside another function: look inside the current function, then where that function was defined, and so on, all the way up to the global environment, and then on to other loaded packages.
     - The same rules apply to closures, functions created by other functions.

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

  - ◇ It works because k preserves the environment in which it was defined and because the environment includes the value of y.
  - ◇ *Environments* gives some pointers on how you can dive in and figure out what values are stored in the environment associated with each function.

2. functions vs. variables
   - ▷ The same principles apply regardless of the type of associated value - finding functions works exactly the same way as finding variables
   - ▷ For functions, there is one small tweak to the rule. If you are using a name in a context where it's obvious that you want a function (e.g. f(3)), R will ignore objects that are not functions while it is searching.

```
n <- function(x) x / 2
o <- function() {
n <- 10 n(n)
}
o()
#> [1] 5
```

   - ▷ However, using the same name for functions and other objects will make for confusing code, and is generally best avoided.
3. a fresh start
   - ▷ exists: it returns TRUE if there's a variable of that name, otherwise it returns FALSE.)

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

   - ▷ It returns the same value, 1, every time. This is because every time a function is called, a new environment is created to host execution. A function has no way to tell what happened the last time it was run; each invocation is completely independent.
4. dynamic lookup
   - ▷ Lexical scoping determines where to look for values, not when to look for them. R looks for values when the function is run, not when it's created. This means that the output of a function can be different depending on objects outside its environment.
   - ▷ It means the function is no longer *self-contained.* This is a common error - if you make a spelling mistake in your code, you won't get an error when you create the function, and you might not even get one when you run the function, depending on what variables are defined in the global environment.
   - ▷ Function findGloabals() in package codetools can list all the external dependencies of a function:
     f <- function() x + 1
     codetools::findGlobals(f)
     #> [1] "+" "x"
   - ▷ We can mannually change the environment of the function to the emptyenv(), an evenironment which contains absolutely nothing:
     environment(f) <- emptyenv()

```
f()
#> Error: could not find function "+"
```

- This doesn't work because R relies on lexical scoping to find everything, even the + operator.
- It's never possible to make a function completely self-contained because you must always rely on functions defined in base R or other packages.

Every operation is a function call

- To understand computations in R, two slogans are helpful:
  - Everything that exists is an object.
  - Everything that happens is a function call.
- Note that ', the backtick, lets you refer to functions or variables that have otherwise reserved or illegal names:

```
x <- 10; y <- 5
'+'(x, y)
#> [1] 15
'if'(i == 1, print("yes!"), print("no."))
'{'(print(1), print(2), print(3))
```

- In the following examples, note the difference b/t '+' and "+". The first one is the value of the object called +, and the second is a string containing the character +. The 2nd works because lapply can be given the name of a function instead of the function itself.

```
sapply(1:5, '+', 3)
#> [1] 4 5 6 7 8
sapply(1:5, "+", 3)
#> [1] 4 5 6 7 8
```

- A more useful application is combining lapply() or sapply() with subsetting:

```
x <- list(1:3, 4:9, 10:12)
sapply(x, "[", 2)
#> [1] 2 5 11
# equivalent to
sapply(x, function(x) x[2])
#> [1] 2 5 11
```

Function argumens

- The formal arguments are a property of the function, whereas the actual or calling arguments can vary each time you call the function.
- **Calling functions**
  - When calling a function you can specify arguments **by position**, **by complete name**, or **by partial name**. Arguments are matched **first by exact name** (perfect matching), **then by prefix matching** and **finally by position**.

- ▷ Named arguments should always come after unnamed arguments.
- ▷ If a function uses ... (discussed in more detail below), you can only specify arguments listed after ... with their full name.

♢ **Calling a function given a list of arguments**

- ▷ Suppose you had a list of function arguments, you need do.call() to send that list to a function:

```
do.call(mean, list(1:10, na.rm = TRUE))
#> [1] 5.5
# Equivalent to
mean(1:10, na.rm = TRUE)
#> [1] 5.5
```

♢ **Default and missing arguments**

- ▷ Function arguments in R can have default values.
- ▷ Since arguments in R are evaluated lazily (more on that below), the default value can be defined in terms of other arguments:

```
g <- function(a = 1, b = a * 2) {c(a, b) }
```

- ▷ Default arguments can even be defined in terms of variables created **within** the function.
- ▷ You can determine if an argument was supplied or not with the missing() function:

```
i <- function(a, b) { c(missing(a), missing(b)) }
i(a = 1)
#> [1] FALSE TRUE
i(1, 2)
#> [1] FALSE FALSE
```

- ▷ We also can set the default value to NULL, and use is.null() to check if the argument was supplied.

♢ **Lazy evaluation**

- ▷ By default, R function arguments are lazy - they're only evaluated if they're actually used:

```
f <- function(x) { 10 }
f(stop("This is an error!"))
#> [1] 10
```

- ▷ If you want to ensure that an argument is evaluated you can use force:

```
f <- function(x) { force(x) 10 }
f(stop("This is an error!"))
#> Error: This is an error!
```

- ▷ This is important when creating closures with lapply or a loop:

```
add <- function(x) { function(y) x + y }
adders <- lapply(1:10, add)
adders[[1]](10)
#> [1] 20
adders[[10]](10)
#> [1] 20
```

x is lazily evaluated in the first time that we call one of the adder functions. At this point, the loop is complete and the final value of x is 10. Therefore all of the adder functions will add 10 on to their input.

```
add <- function(x) { force(x) function(y) x + y }
adders2 <- lapply(1:10, add)
adders2[[1]](10)
#> [1] 11
adders2[[10]](10)
#> [1] 20
```

  - ◇ the force function is defined as force <- function(x) x.

- ▷ Default arguments are evaluated inside the function. This means that if the expression depends on the current environment the results will differ depending on whether you use the default value or explicitly provide one.

```
f <- function(x = ls()) { a <- 1 x }
# ls() evaluated inside f:
f()
#> [1] "a" "x"
# ls() evaluated in global environment:
f(ls())
```

- ▷ More technically, an unevaluated argument is called a **promise**, or (less commonly) a **thunk**. A promise is made up of two parts:
  - ◇ the expression which gives rise to the delayed computation. (It can be accessed with substitute.)
  - ◇ the environment where the expression was created and where it should be evaluated.
- ▷ Find more information about a promise using pryr::promise_info.
- ▷ Laziness is useful in if statements - the second statement below will be **evaluated only** if the first is true.

```
x <- NULL
if (!is.null(x) && x > 0) {
}
```

- ▷ Sometimes you can also use laziness to eliminate an if statement altogether. For example, instead of:

```
if (is.null(a)) stop("a is null")
#> Error: a is null
```

You could write:

```
!is.null(a) || stop("a is null")
#> Error: a is null
```

♢ ...

- ▷ Special argument called ..., this argument will match any arguments not otherwise matched, and can be easily passed on to other functions.
- ▷ This is useful if you want to collect arguments to call another function, but you don't want to prespecify their possible names.

- ... is often used in conjunction with S3 generic functions to allow individual methods to be more flexible.
- the advantages and disadvantages of ...: it makes plot() very flexible, but to understand how to use it, we have to carefully read the documentation.
- To capture ... in a form that is easier to work with, you can use list(...).
- Using ... comes at a price - any misspelled arguments will not raise an error, and any arguments after ... must be fully named. This makes it easy for typos to go unnoticed:

```
sum(1, 2, na.mr = TRUE)
#> [1] 4
```

Special calls

◇ **Infix functions**

- Most functions in R are "**prefix**" operators: the name of the function comes before the arguments.
- **infix** functions where the function name comes in between its arguments, like + or -.
- All user created infix functions names must start and end with % and R comes with the following infix functions predefined: %%, %*%, %/%, %in%, %o%, %x%.
- The complete list of built-in infix operators that don't need % is: ::, :::, $, @, ^, *, /, +, -, >, >=, <, <=, ==, !=, !, &, &&, |, ||, ~, <-, <<-
- Note that when creating the function, you have to put the name in quotes because it's a special name. This is just a syntactic sugar for an ordinary function call.
- as far as R is concerned there is no difference between these two expressions:

```
"new" %+% " string"
#> [1] "new string"
`%+%`("new", " string")
#> [1] "new string"
```

- The names of infix functions are more flexible than regular R functions: they can contain any sequence of characters (except "%", of course)
- R's default precedence rules mean that infix operators are composed from left to right:

```
"%-%" <- function(a, b) paste0("(", a, " %-% ", b, ")")
"a" %-% "b" %-% "c"
#> [1] "((a %-% b) %-% c)"
```

- A way of providing a default value in case the output of another function is NULL (author defined function):

```
"%||%" <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

◇ **Replacement functions**

- Replacement functions act *like* they modify their arguments in place (because they actually create a modified copy[2]), and have the special name xxx<-
- They typically have two arguments (x and value), although they can have more, and they must return the modified object.

[2]We can see that by using pryr::address() to find the memory address of the underlying object. Built in functions that are implemented using .Primitive will modify in place

- ▷ Example:
  ```
  "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
  ```
  When R evaluates the assignment second(x) <- 5, it notices that the left hand side of the <- is not a simple name, so it looks for a function named second<- to do the replacement.
- ▷ Combining replacement and subsetting:
  ```
  x <- c(a = 1, b = 2, c = 3)
  names(x)
  #> [1] "a" "b" "c"
  names(x)[2] <- "two"
  names(x)
  #> [1] "a" "two" "c"
  ```
  This is done as follows in R:
  ```
  ‘*tmp*‘ <- names(x)
  ‘*tmp*‘[2] <- "two"
  names(x) <- ‘*tmp*‘
  ```
- ▷ setdiff
  ```
  function (x, y) {
  x <- as.vector(x)
  y <- as.vector(y)
  unique(if (length(x) || length(y))
  x[match(x, y, 0L) == 0L]
  else x)
  }
  ```

Return values

- ◇ The last expression evaluated in a function becomes the return value, the result of invoking the function.
- ◇ it's good style to reserve the use of an explicit return() for when you are returning early, such as for an error, or a simple case of the function.
- ◇ Functions can return only a single value. In practice, this is not a limitation because you can always return a list containing any number of objects.
- ◇ The functions that are the easiest to understand and reason about are **pure functions**: functions that always map the same input to the same output and have no other impact on the workspace.
- ◇ In other words, pure functions have **no side-effects**: they don't affect the state of the world in any way apart from the value they return.
- ◇ R protects you from one type of side-effect: most R objects have copy-on-modify semantics. So modifying a function argument does not change the original value[3]:

[3] There are two important exceptions to the copy-on-modify rule: environments and reference classes. These can be modified in place, so extra care is needed when working with them.

```
f <- function(x) { x$a <- 2 x }
x <- list(a = 1)
f(x)
#> $a
#> [1] 2
x$a
#> [1] 1
```

- Note that the performance consequences are a result of R's implementation of copy-on-modify semantics, they are not true in general.
- Most base R functions are pure, with a few notable exceptions:
  - library which loads a package, and hence modifies the search path.
  - setwd, Sys.setenv, Sys.setlocale which change the working directory, environment variables and the locale respectively.
  - plot and friends which produce graphical output.
  - write, write.csv, saveRDS etc. which save output to disk.
  - options and par which modify global settings.
  - S4 related functions which modify global tables of classes and methods.
  - Random number generators which produce different numbers each time you run them.
- Functions can return invisible values, which are not printed out by default when you call the function.

```
f2 <- function() invisible(1)
f2() == 1
#> [1] TRUE
```

- You can always force an invisible value to be displayed by wrapping it in parentheses.
- on.exit()
  - Another more casual way of cleaning up is the on.exit function, which is called when the function terminates. It's not as fine grained as tryCatch, but it's a bit less typing.

```
in_dir <- function(path, code) {
cur_dir <- getwd()
on.exit(setwd(cur_dir))
force(code) }
```

2.5 OO (Object Oriented) field guide

Introduction

- Central to any object-oriented system are the concepts of class and method.
- A **class** defines the behaviour of objects by describing their attributes and their relationship to other classes.
- The class is also used when selecting **methods**, (i.e.) functions that behave differently depending on the class of their input.

- ♢ Classes are usually organised in a **hierarchy**: if a method does not exist for a child, then the parent's method is used instead. This means the child **inherits** behaviour from the parent.
- ♢ R's three OO systems differ in how classes and methods are defined:
  - ▷ **S3** implements a style of OO programming called *generic-function* OO, in contrast to *message-passing* OO used in Java, C++ and C#. In S3, while computations are still carried out via methods, a special type of function called a **generic function** decides which method to call. S3 is a very casual system, it has not formal definition of classes.
  - ▷ **S4** works similarly to S3, but is more formal. Two major differences to S3: S4 has formal class definitions, which describe the representation and inheritance of each class, and has special helper functions for defining generics and methods; S4 also has multiple dispatch, which means that generic functions can pick methods based on the class of any number of arguments, not just one.
  - ▷ **Reference classes** (RC) are quite different from S4 and S3. RC implements message-passing OO, so methods belong to classes, not functions. `$` is used to separate objects and methods, so method calls look like `canvas$drawRect("blue")`. RC objects are also mutable: they don't use R's usual copy-on-modify semantics, but are modified in place.
  - ▷ **Base types**, the internal C-level types that underlie the other OO systems. Base types are mostly manipulated using C code, but they're important to know about because they provide the building blocks for the other OO systems.

Base types

- ♢ Underlying every R object is a C structure (or struct) that describes how that object is stored in memory.
- ♢ The struct includes the contents of the object, the information needed for memory management, and most importantly for this section, a **type**. This is the **base type** of an R object.
- ♢ Base types are not really an object system. In fact, only the R core team can create new types.
- ♢ Most common base types are atomic vectors and lists, other base types encompass functions, environments and other more exotic objects likes names, calls and promises.
- ♢ To determine an object's base type, use `typeof()`.
- ♢ Be aware that the names of base types are not used consistently throughout R: the type and the corresponding "is" function may have different names: for example, the type of a function is "closure" and the type of the primitive function is "builtin".[4]
- ♢ Functions that behave differently for different base types are almost always written in C, where dispatch occurs using switch statements (e.g. `switch(TYPEOF(x))`).
- ♢ S3 objects can be built on top of any base type, S4 objects use a special base type, and RC objects are a combination of S4 and environments (another base type).
- ♢ To see if an object is a pure base type, i.e. it doesn't also have S3, S4 or RC behaviour, check that `is.object(x)` returns `FALSE`.

[4]corresponding to `is.function()` and `is.primitive()`.

S3

- ♢ It is the only OO system used in the base and stats packages, and it's the most commonly used system in packages.
- ♢ S3 is informal and ad hoc, but it has a certain elegance in its minimalism: you couldn't take any part of it away and still have a useful OO system.
- ♢ **Recognising objects, generic functions and methods**
  - ▷ You can check this with `is.object(x) & !isS4(x)` (the second expression tests whether it's not an S4 object).
  - ▷ An easier way to determine the OO system of an object is to use `pryr::otype()`
  - ▷ In S3, methods are associated with functions, called **generic functions** or **generics** for short, not objects or classes. [5]
  - ▷ To determine if a function is an S3 generic, you can look at its source code for a call to `UseMethod()`: that's the function that figures out the correct method to call, the process of **method dispatch**.
  - ▷ pryr also provides `ftype()` which describes the object system, if any, associated with a function: `ftype(mean)` *#> [1] "s3" "generic"*
  - ▷ Some S3 generics, like `[`, `sum` and `cbind`, don't call `UseMethod()` because they are implemented in C. Instead, they call the C functions `DispatchGroup()` or `DispatchOrEval()`. These function that do method dispathc in C code are called **internal generics**.
  - ▷ Internal generics are documented in `?"internal generic"`. `ftype()` knows about these special cases too.
  - ▷ The job of an S3 generic is to call an S3 method specialised for a given class.
  - ▷ Format of the names for S3 methods is like `generic.class()`, e.g. `print.factor()` or `mean.Date()`, etc.
  - ▷ `pryr::ftype()` knows about these exceptions, so you can use it to figure out if a function is an S3 method or generic.
  - ▷ You can see all the methods of a generic using the `methods()` function: `methods(mean)`[6]
  - ▷ You can also list all generics that have a method for a given class: `methods(class = "ts")`[7]
- ♢ **Defining classes and creating objects**
  - ▷ To make an object an instance of a class, you just take an existing base object and set the class attribute.
  - ▷ You can do that during creation with `structure()`, or after the fact with `attr<-()`. However, if you're modifying an existing object, using `class<-()` will more clearly communicate your intent.
  - ▷ S3 objects are usually built on top of lists, or atomic vectors with attributes.
  - ▷ You can determine the class of any object using `class(x)`, and see if an object inherits from a specific class using `inherits(x, "classname")`.
  - ▷ The class of an S3 object can be a vector (i.e. belongs to more than one class), which describes behaviour from most to least specific. For example, the class of the `glm()` object is `c("glm", "lm")` indicating that generalised linear models inherit behaviour from linear models.

[5]This is different from most of other programming languages.
[6]Apart from methods defined in the base package, most S3 methods will not be visible: use getS3method() to read their source code.
[7]Because there's no central repository, there's no way to list all S3 classes.

- ▷ Most S3 classes provide a constructor function. You should use it if it's available (like for factor() and data.frame()). This ensures that you're creating the class with the correct components. Constructor functions usually have the same name as the class.

```
foo <- function(x) {
if (!is.numeric(x)) stop("X must be numeric")
structure(list(x), class = "foo")
}
```

- ▷ Apart from developer supplied constructor functions, S3 has no checks for correctness. This means you can change the class of existing objects.[8]

♢ Creating new methods and generics

- ▷ To add a new generic, create a function that calls UseMethod().
- ▷ UseMethod() takes two arguments: the name of the generic function, and the argument to use for method dispatch. If you omit the second argument it will dispatch on the first argument to the function.
- ▷ There's no need to pass any of the arguments of the generic to UseMethod(). In fact, you shouldn't do so. UseMethod() uses black magic to find them out for itself.

```
f <- function(x) UseMethod("f")
```

- ▷ A generic isn't useful without some methods. To add a method, you just create a regular function with the correct (generic.class) name:

```
f.a <- function(x) "Class a"
```

♢ Method dispatch

- ▷ UseMethod() creates a vector of function names, like paste0("generic", ".", c(class(x), "default")) and looks for each in turn. The "default" class makes it possible to set up a fall back method for otherwise unknown classes:

```
f <- function(x) UseMethod("f")
f.a <- function(x) "Class a"
f.default <- function(x) "Unknown class"
f(structure(list(), class = "a"))
#> [1] "Class a"
# No method for b class, so uses method for a class
f(structure(list(), class = c("b", "a")))
#> [1] "Class a"
# No method for c class, so falls back to default
f(structure(list(), class = "c"))
#> [1] "Unknown class"
```

- ▷ Group generics make it possible to implement methods for multiple generics with one function.
- ▷ The four group generics and the functions they include are:
  - ⋄ Math: abs, sign, sqrt, floor, cos, sin, log, exp, ...
  - ⋄ Ops: +, -, *, /, ^, %%, %/%, &, |, !, ==, !=, <, <=, >=, >
  - ⋄ Summary: all, any, sum, prod, min, max, range

[8]While you can change the type of an object, you never should. R doesn't protect you from yourself: you can easily shoot yourself in the foot. As long as you don't aim the gun at your foot and pull the trigger, you won't have a problem.

    - Complex: Arg, Conj, Im, Mod, Re
  - Find out more about them in ?groupGeneric.
  - The most important thing to take away from this is to recognise that Math, Ops, Summary and Complex aren't real functions. They represent groups of functions. Note that inside a group generic function a special variable .Generic provides the actual generic function called.
  - If you have complex class hierarchies it's sometimes useful to call the "parent" method.[9]
  - Because methods are normal R functions, you can call them directly. (i.e type the full name as generic.class). However, this is just as dangerous as changing the class of an object.
  - You can also call an S3 generic with a non-S3 object. Non-internal S3 generics will dispatch on the **implicit class** of base types. (Internal generics don't do that for performance reasons.)
- Carefully read ?"internal generic" . UseMethod()

S4

- S4 works in a similar way to S3, but it adds formality and rigour. Methods still belong to functions, not classes, but:
  - Classes have a formal definition, describing their fields and inheritance structure (parent classes).
  - Method dispatch can be based on multiple arguments to a generic function, not just one.
  - There is a special operator, @, for extracting fields (aka **slots**) out of an S4 object.
- All S4 related code is stored in the methods package. This package is always available when you're running R interactively, but is not always loaded automatically when running R in batch mode. For this reason, it's a good idea to include an explicit library(methods) whenever you're using S4.
- Good references:
  - S4 system development in Bioconductorhttp://www.bioconductor.org/help/course-materials/2010/AdvancedR/S4InBioconductor.pdf
  - Book: Software for Data Analysis (by John Chambers) http://ishare.iask.sina.com.cn/f/12201138.html
  - Stackoverflow answers to S4 questions by Martin Morgan.http://stackoverflow.com/search?tab=votes&q=user%3a547331%20%5bs4%5d%20is%3aanswe
- **Recognising objects, generic functions and methods**
  - You can identify an S4 object because str() describes it as a "**formal**" class, isS4() is TRUE, and pryr::otype() returns "S4".
  - There aren't any S4 classes in the commonly used base packages (stats, graphics, utils, datasets, and base).
  - You can determine the class of an S4 object with class() and test if an object inherits from a specific class with is().
  - You can get a list of all S4 generics with getGenerics(), and a list of all S4 classes with getClasses(), but note that this list includes shim classes for S3 classes and base types.
  - You can list all S4 methods with showMethods(), optionally restricting either by generic or by class (or both). It's also a good idea to supply where = search() to restrict to methods available from the global environment.

[9]This is an advanced technique: read about it in ?NextMethod.

◇ Defining classes and creating objects

- ▷ In setting the class for an object in S4 is much stricter than in S3: you must define the representation of the class using setClass(), and create an new object with new(). You can find the documentation for a class with a special syntax: class?className, e.g. class?mle.
- ▷ An S4 class has three key properties:
  - ◇ A **name**: an alpha-numeric class identifier. S4 class names should use **UpperCamelCase**.
  - ◇ A named list of **slots** (fields), providing slot names and permitted classes. For example, a person class might be represented by a character name and a numeric age: list(name = "character", age = "numeric").
  - ◇ A string giving the class it inherits from, or in S4 terminology, that it **contains**. You can provide multiple classes for multiple inheritance, but this is an advanced technique and it adds much complexity.
- ▷ In slots and contains you can use S4 classes, S3 classes registered with setOldClass(), or the implicit class of a base type. In slots you can also use the special class ANY which does not restrict the input.
- ▷ S4 classes have other optional properties like a validity method that tests if an object is valid, and a prototype that defines slot default values.
- ▷ See ?setClass for more details.
- ▷ The define of the new S4 class is similar to other language like Python.
- ▷ Example: create two classes "Person" and "Employee", where "Employee" inherits the slots and methods from "Person".

```
setClass("Person",
slots = list(name = "character", age = "numeric"))
setClass("Employee",
slots = list(boss = "Person"),
contains = "Person")
alice <- new("Person", name = "Alice", age = 40)
john <- new("Employee", name = "John", age = 20, boss = alice)
```

- ▷ Most S4 classes also come with a **constructor function** with the same name as the class: if that exists, use it instead of calling new() directly.
- ▷ To access slots of an S4 object you use @ or slot() [10]

```
alice@age
#> [1] 40
slot(john, "boss")
#> An object of class "Person"
#> Slot "name":
#> [1] "Alice"
#>
#> Slot "age":
#> [1] 40
```

[10] @ is equivalent to $, and slot() to [[.

- ▷ If an S4 object contains (inherits) from an S3 class or a base type, it will have a special .Data slot which contains the underlying base type or S3 object.

```
setClass("RangedNumeric",
contains = "numeric",
slots = list(min = "numeric", max = "numeric"))
rn <- new("RangedNumeric", 1:10, min = 1, max = 10)
rn@min
#> [1] 1
rn@.Data
#> [1] 1 2 3 4 5 6 7 8 9 10
```

- ▷ When you're interactively experimenting with S4, if you modify a class, make sure you also recreate any objects of that class, otherwise you'll end up with invalid objects.

◇ **Creating new methods and generics**

- ▷ S4 provides special functions for creating new generics and methods.
- ▷ setGeneric() will create a new generic or convert an existing function into a generic.
- ▷ setMethod() takes the name of the generic, the classes the method should be associated with and a function that implements the method.

```
setGeneric("union")
#> [1] "union"
setMethod("union", c(x = "data.frame", y = "data.frame"),
function(x, y) { unique(rbind(x, y))
}
)
#> [1] "union"
```

- ▷ If you create a new generic from scratch, you also need to supply a function that calls standard-Generic():

```
setGeneric("myGeneric", function(x) {
standardGeneric("myGeneric")
})
#> [1] "myGeneric"
```

◇ **Method dispatch**

- ▷ S4 method dispatch is the same as S3 dispatch if your classes only inherit from a single parent, and you only dispatch on one class.
- ▷ The main difference is how you set up default values: S4 uses the special class ANY to match any class and "missing" to match a missing argument.
- ▷ Like S3, S4 also has group generics, documented in ?S4groupGeneric, and a way to call the "parent" method, callNextMethod().
- ▷ If you dispatch on multiple arguments, or your classes use multiple inheritance, things can be very complicated. The rules are described in ?Methods, but they are complicated and it's difficult to predict which method will be called.[11]

[11]The author strongly suggest avoiding multiple inheritance and mutilple dispatch unless absolutely necessary.

- ▷ There are two functions that identify the method that will be invoked given the specification of a call to a generic:
  - ◇ ```
    # From methods: takes generic name and class names
    selectMethod("nobs", list("mle"))
    ```
  - ◇ ```
    # From pryr: takes an unevaluated function call
    method_from_call(nobs(fit))
    ```
- ▷ Read ?Classes for an overview of S4 classes.
- ▷ What happens if you pass an S4 object to an S3 generic? What happens if you pass an S3 object to an S4 generic?
- ▷ Read ?setOldClass: Registering via setOldClass allows S3 classes to appear in method signatures, as a slot in an S4 class, or as a superclass of an S4 class.

RC (Reference Classes)

- ♢ The newest OO system in R. They were introduced in version 2.12.
- ♢ They are fundamentally different to S3 and S4 because:
  - ▷ RC methods belong to objects, not functions
  - ▷ RC objects are mutable: the usual R copy-on-modify semantics do not apply
  - ▷ These properties make RC objects behave more like objects do in most other programming languages, e.g., Python, Ruby, Java and C#.
  - ▷ Surprisingly, reference classes are implemented in R, not C: they are a special S4 class that wraps around an environment.
- ♢ **Defining classes and creating objects**
  - ▷ RC classes are best used for describing stateful objects, objects that change over time, so we'll create a simple class to model a bank account.
  - ▷ Creating a new RC class is similar to creating a new S4 class, but you use setRefClass() instead of setClass().
  - ▷ The first, and only required argument, is an alpha-numeric **name**.
  - ▷ While you can use new() to create new RC objects, it's good style to use the object returned by setRefClass() to generate new objects.
    ```
    Account <- setRefClass("Account")
    Account$new()
    #> Reference class object of class "Account"
    ```
  - ▷ setRefClass() also accepts a list of name-class pairs that define class **fields** (equivalent to S4 slots). Additional named arguments passed to **new()** will set initial values of the fields. You can get and set field values with **$:**
    ```
    Account <- setRefClass("Account",
    fields = list(balance = "numeric"))
    a <- Account$new(balance = 100)
    a$balance
    #> [1] 100
    a$balance <- 200
    a$balance
    #> [1] 200
    ```

- Instead of supplying a class name for the field, you can provide a single argument function which will act as an accessor method. This allows you to add custom behaviour when getting or setting a field. See ?setRefClass for more details.
- Note that RC objects are **mutable**, i.e., they have reference semantics, and are not copied-on-modify:

  ```
  b <- a b$balance
  #> [1] 200
  a$balance <- 0
  b$balance
  #> [1] 0
  ```
- For this reason, RC objects come with a copy() method that allow you to make a copy of the object:

  ```
  c <- a$copy()
  c$balance
  #> [1] 0
  a$balance <- 100
  c$balance
  #> [1] 0
  ```
- RC methods are associated with a class and can modify its fields in place. In the following example, note that you access the value of fields with their name, and modify them with <<-.

  ```
  Account <- setRefClass("Account",
  fields = list(balance = "numeric"),
  methods = list(
  withdraw = function(x) {
  balance <<- balance - x
  },
  deposit = function(x) {
  balance <<- balance + x
  }
  )
  )
  a <- Account$new(balance = 100)
  a$deposit(100)
  a$balance
  #> [1] 200
  ```
- The final important argument to setRefClass() is contains. This is the name of the parent RC class to inherit behaviour from.

  ```
  NoOverdraft <- setRefClass("NoOverdraft",
  contains = "Account",
  methods = list(
  withdraw = function(x) {
  if (balance < x) stop("Not enough money")
  balance <<- balance - x
  }
  ```

```
)
)
accountJohn <- NoOverdraft$new(balance = 100)
accountJohn$deposit(50)
accountJohn$balance
#> [1] 150
accountJohn$withdraw(200)
#> Error: Not enough money
```

- ▷ All reference classes eventually inherit from envRefClass. It provides useful methods like copy() (shown above), callSuper() (to call the parent field), field() (to get the value of a field given its name), export() (equivalent to as) and show() (overridden to control printing).[12]

- ◇ **Recognising objects and methods**
  - ▷ You can recognise RC objects because they are S4 objects (isS4(x)) that inherit from "refClass" (is(x, "refClass")). pryr::otype() will return "RC". RC methods are also S4 objects, with class refMethodDef.

- ◇ **Method dispatch**
  - ▷ Method dispatch is very simple in RC because methods are associated with classes, not functions. When you call x$f(), R will look for a method f in the class of x, then in its parent, then its parent's parent, and so on. From within a method, you can call the parent method directly with callSuper(...).

- ◇ Exercises questions.

Picking a system

- ◇ S3 is a little quirky, but it gets the job done with a minimum of code.
- ◇ If you are creating more complicated systems of interrelated objects, S4 may be more appropriate.
- ◇ If you've mastered S3, S4 is relatively easy to pick up: the ideas are all the same, it is just more formal, more strict and more verbose.
- ◇ Resources to learn S4
  - ▷ Bioconductor package: https://www.google.com/search?q=bioconductor+s4
  - ▷ Matrix package: vignette("Intro2Matrix", package = "Matrix")
- ◇ If you've programmed in mainstream OO language, RC will seem very natural. But because they can introduce side-effects through mutable state, they are harder to understand
- ◇ Generally, when using RC objects you want to minimise side effects as much as possible, and use them only where mutable states are absolutely required. The majority of functions should still be "functional", and free of side effects.

[12]See the inheritance section in setRefClass() for more details.

2.6 Environments

Introduction

- ♢ Understanding environments is important to understanding scoping.
- ♢ what an environment is and how to inspect and manipulate them
- ♢ the four types of environments associated with a function
- ♢ how to work in an environment outside of a function with local()
- ♢ the four ways of binding names to values in an environment

Environment basics

- ♢ **What is an environment?**
  - ▷ The job of an environment is to associate, or bind, a set of names to a set of values.
  - ▷ Environments are the data structures that power scoping.
  - ▷ An environment is very similar to a list, with three important exceptions:
    - ⋄ Environments have reference semantics. So R's usual copy on modify rules do not apply. Whenever you modify an environment, you modify every copy.
    - ⋄ Environments have parents. If an object is not found in an environment, then R will look at its parent (and so on).
    - ⋄ There is only one exception: the **empty** environment does not have a parent.[13]
    - ⋄ Every object in an environment must have a name. And, those names must be unique.
  - ▷ Technically, an environment is made up of a **frame**, a collection of named objects (like a list), and a reference to a parent environment.
  - ▷ As well as powering scoping, environments can also be useful data structures because they have reference semantics and can work like a **hashtable**.
- ♢ **Manipulatin gand inspecting environments**
  - ▷ You can create environments with new.env(), see their contents with ls(), and inspect their parent with parent.env().
  - ▷ You can modify environments in the same way you modify lists:

```
ls(e)
#> [1] "a"
e$a <- 1
ls(e)
#> [1] "a"
e$a
#> [1] 1
```

  - ▷ By default ls only shows names that don't begin with .. Use all.names = TRUE (or all for short) to show all bindings in an environment.

13

· It's rare to talk about the children of an environment because there are no back links: given an environment we have no way to find its children.

- ▷ Another useful technique to view an environment is to coerce it to a list: as.list(env)
- ▷ You can extract elements of an environment using $ or [[, or get(). While $ and [[will only look within an environment, get, using the regular scoping rules, will also look in the parent if needed. $ and [[will return NULL if the name is not found, while get returns an error.
- ▷ Deleting objects from environments works a little differently from lists. With a list you can remove an entry by setting it to NULL. Working in environments, instead you need to use rm(). E.g. rm("a", envir = e)
- ▷ Generally, when you create your own environment, you want to manually set the parent environment to the empty environment. This ensures you don't accidentally inherit objects from somewhere else:

```
x <- 1
e1 <- new.env()
get("x", e1)
#> [1] 1
e2 <- new.env(parent = emptyenv())
get("x", e2)
#> Error: object 'x' not found
```

- ▷ You can determine if a binding exists in a environment with the exists() function. Like get(), the default is to follow regular scoping rules and look in parent environments. If you don't want this behavior, use inherits = FALSE:

```
exists("x", e1)
#> [1] TRUE
exists("x", e1, inherits = FALSE)
#> [1] FALSE
```

♢ Special environments

- ▷ There are a few special environments that you can access directly:
  - ◇ globalenv(): the user's workspace (top-level), the most common one;[14]
  - ◇ baseenv(): the environment of the base package, its parent is the empty environment.
  - ◇ emptyenv(): the ultimate ancestor of all environments, the only environment without a parent.
- ▷ search() lists all environments between and including the global and base environments.
  - ◇ This is called the search path because any object in these environments can be found from the top-level interactive workspace.
  - ◇ It contains an environment for each loaded package and for each object (environment, list or Rdata file) that you've attach()ed.
  - ◇ It also contains a special environment called Autoloads which is used to save memory by only loading package objects (like big datasets) when needed.
- ▷ You can access the environments of any environment on the search list using as.environment().

♢ Where

- ▷ pryr::where is built to tell us the environment where a variable is located. E.g. where("mean") #> <environment: base>

[14]The parent of the global environment is one of the packages you have loaded (the exact order will depend on the order in which packages were loaded).

- where() obeys the regular rules of variable scoping, but instead of returning the value associated with a name, it returns the environment in which it was defined.
- where

```
#> function (name, env = parent.frame())
#> {
#> stopifnot(is.character(name), length(name) == 1)
#> env <- to_env(env)
#> if (identical(env, emptyenv())) {
#> stop("Can't find ", name, call. = FALSE)
#> }
#> if (exists(name, env, inherits = FALSE)) {
#> env
#> }
#> else {
#> where(name, parent.env(env))
#> }
#> }
#> <environment: namespace:pryr>
```
[15]

- There are three main components:
  - the base case (what happens when we've recursed up to the empty environment)
  - a Boolean that determines if we've found what we wanted[16]
  - the recursive statement that re-calls the function using the parent of the current environment.
- Write your own version of get() using a function written in the style of where().

```
get2<-function(name,env=parent.frame()){
stopifnot(is.character(name),length(name)==1)
if (identical(env,emptyenv())){
stop ("Can't find",name,fall.=F)
}
if (exists(name,env,inherits=F)){
env[[name]]
}
else {
get2(name,parent.env(env))
}
}
```

Function environments

◇ The environment where the function is created

- Most of the time, you do not create environments directly. They are created as a consequence of working with functions.

[15] It's natural to work with environments recursively, so we'll see this style of function structure frequently.

[16] Note that to check if the environment is the same as the empty environment, we need to use identical(). Unlike the element-wise ==, this performs a whole object comparison.

- ▷ When a function is created, it gains a reference to the environment where it was made. This is the parent, or enclosing, environment of the function used by lexical scoping.

  You can access this environment with the environment() function: environment(plot)
  #> <environment: namespace:graphics>
- ▷ The enclosing environment is particularly important for closures:

```
plus <- function(x) {
function(y) x + y
}
plus_one <- plus(1)
plus_one(10)
#> [1] 11
plus_two <- plus(2)
plus_two(10)
#> [1] 12
environment(plus_one)
#> <environment: 0x106f1e788>
parent.env(environment(plus_one))
#> <environment: R_GlobalEnv>
```

♢ The environment where the function resides

- ▷ The environment of a function (use funenv() to check) and the environment where it resides (use where() to check) might be different.
- ▷ The environment where the function lives determines how we find the function.
- ▷ The environment of the function determines how we find values inside the function.
- ▷ This important distinction is what enables package namespaces to work.
- ▷ The package environment contains only functions and objects that should be visible to the user, but the namespace environment contains both internal and external functions.
- ▷ This mechanism makes it possible for packages to have internal objects that can be accessed by its functions, but not by external functions.

♢ The environment created when a function is run

- ▷ Example:

```
f <- function(x) {
if (!exists("a", inherits = FALSE)) {
message("Defining a")
a <- 1
} else {
a <- a + 1
}
a
}
```

  It will return the same value each and every time it is called. This is because each time a function is called, a new environment is created to host execution.

- ▷ Calling environment() with no arguments returns the environment in which the call was made, so we can use it to confirm that functions have new hosting environments at every invocation.

```
f <- function(x) {
list(
e = environment(),
p = parent.env(environment())
)
}
str(f())
#> List of 2
#> $ e:<environment: 0x10528b5f0>
#> $ p:<environment: R_GlobalEnv>
```

◇ **The environment where the function is called**

- ▷ Example

```
f <- function() {
x <- 10
function() {
x
}
}
g <- f()
x <- 20
g()
#[1] 10
```

  - ◇ The top-level x is a red herring: using the regular scoping rules, g() looks first where it is defined and finds the value of x is 10.[17]
  - ◇ However, it is still meaningful to ask what value x is associated with in the environment where g() is called. x is 10 in the environment where g() is defined, but it is 20 in the environment where g() is called.
- ▷ We can access this environment using the confusingly named parent.frame(). This function returns the environment where the function is called.

```
f2 <- function() {
x <- 10
function() {
def <- get("x", environment())
cll <- get("x", parent.frame())
list(defined = def, called = cll)
}
}
g2 <- f2()
x <- 20
str(g2())
```

[17] I.e. in lexical scoping, function looks for the object in the defining environment first.

```
#> List of 2
#> $ defined: num 10
#> $ called : num 20
```

▷ We can get a list of all calling environments using sys.frames():[18]

```
x <- 0
y <- 10
f <- function(x) {
x <- 1
g(x)
}
g <- function(x) {
x <- 2
h(x)
}
h <- function(x) {
x <- 3
i(x)
}
i <- function(x) {
x <- 4
sys.frames()
}
es <- f()
sapply(es, function(e) get("x", e, inherits = TRUE))
# [1] 1 2 3 4
sapply(es, function(e) get("y", e, inherits = TRUE))
# [1] 10 10 10 10
```

▷ There are two separate strands of parents when a function is called: calling environments and enclosing environments. Each calling environment will also have a stack of enclosing environments.

▷ Note that a called function has both a stack of called environments and a stack of enclosing environments. However, an environment (or a function object) has only a stack of enclosing environments.

▷ Looking up variables in the calling environment rather than in the defining argument is called **dynamic scoping**.[19]

▷ Dynamic scoping is primarily useful for developing functions that aid interactive data analysis.

Explicit scoping with local

◇ Sometimes it's useful to be able to create a new scope without embedding inside a function. The local function allows you to do exactly that.

```
df <- local({
```

[18]In more complicated scenarios, there's not just one parent call, but a sequence of calls which lead all the way back to the initiating function, called from the top-level.

[19]Few languages implement dynamic scoping (Emacs Lisp is a notable exception). This is because dynamic scoping makes it much harder to reason about how a function operates: not only do you need to know how it was defined, you also need to know in what context it was called.

```
x <- 1:10
y <- runif(10)
data.frame(x = x, y = y)
})
```

is equivalent to:

```
df <- (function() {
x <- 1:10
y <- runif(10)
data.frame(x = x, y = y)
})()
```
[20]

◊ local has relatively limited uses (typically because most of the time scoping is best accomplished using R's regular function based rules) but it can be useful in conjunction with <<-.

Example:

```
a <- 10
my_get <- NULL
my_set <- NULL
local({
a <- 1
my_get <<- function() {
a }
my_set <<- function(value) {
a <<- value
}
})
my_get()
#> [1] 1
my_set(20)
a
#> [1] 10
my_get()
#> [1] 20
```

◊ However, it can be easier to see what's going on if you avoid the implicit environment and create and access it explicitly:

```
my_env <- new.env(parent = emptyenv())
my_env$a <- 1
my_get <- function() {
my_env$a
```

[20] It's the immediately invoked function expression (IIFE). It's used extensively by most JavaScript libraries to avoid polluting the global namespace.

```
}
my_set <- function(value) {
my_env$a <- value
}
```

Assignment: binding names to values

◇ **Definition:** Assignment is the act of binding (or rebinding) a name to a value in an environment.

- ▷ It is the counterpart to scoping, the set of rules that determines how to find the value associated with a name.
- ▷ In R you can not only bind values to names, but you can also bind expressions (promises) or even functions, so that every time you access the value associated with a name, you get something different!

◇ Four main ways of binding names to values in R:

1. With the regular behaviour, `name <- value`, the name is immediately associated with the value in the current environment.
   `assign("name", value)` works similarly, but allows assignment in any environment.
2. The double arrow, `name <<- value`, assigns in a similar way to variable lookup, so that `i <<- i + 1` modifies the binding of the original `i`, which is not necessarily in the current environment.
3. Lazy assignment, `delayedAssign("name", expression)`, binds an expression that isn't evaluated until you look up the name.
4. Active assignment, `makeActiveBinding("name", function, environment)` binds the name to a function, so it is "active" and can return a different value each time the name is found.

◇ **Regular binding**

- ▷ Regular assignment immediately creates a binding between a name and a value in the current environment.
- ▷ There are two types of names: **syntactic** and **non-syntactic**.
  - ◇ Generally, syntactic names consist of letters, digits, . and _, and must start with a letter or . not followed by a number (so .a and ._ are syntactic but .1 is not).[21]
  - ◇ A syntactic name can be used on the left hand side of <-.
  - ◇ However, a name can actually be any sequence of characters; if it's non-syntactic you just need to do a little more work:
    ```
    `a + b` <- 3
    `:)` <- "smile"
    ` ` <- "spaces"
    ls()
    # [1] " " ":)" "a + b"
    ```
- ▷ <- creates a binding in the current environment. There are three techniques to create a binding in another environment:
  1. treating an environment like a list
     ```
     e <- new.env()
     e$a <- 1
     ```

[21]There are also a number of reserved words (e.g. TRUE, NULL, if, function, see make.names()).

2. use assign(), which has three important arguments: the name, the value, and the environment in which to create the binding
   ```
   e <- new.env()
   assign("a", 1, envir = e)
   ```
3. evaluate <- inside the environment.
   ```
   e <- new.env()
   eval(quote(a <- 1), e)
   ```
   or
   ```
   evalq(a <- 1, e)
   ```

▷ **Constants:** Variables whose values can not be changed; they can only be bound once, and never re-dound.

- ◇ We can simulate constants in R using lockBinding, or the infix %<c-% found in pryr:
  ```
  x <- 10
  lockBinding(as.name("x"), globalenv())
  x <- 15
  #> Error: no binding for "x"
  ```
  or
  ```
  x %<c-% 20
  x <- 30
  #> Error: cannot change value of locked binding for 'x'
  ```
- ◇ lockBinding() is used to prevent you from modifying objects inside packages:
  ```
  assign("mean", function(x) sum(x) / length(x), env = baseenv())
  #> Error: cannot change value of locked binding for 'mean'
  ```

♢ <<-

▷ The regular assignment arrow, <-, always creates a variable in the current environment. The special assignment arrow, <<-, never creates a variable in the current environment, but instead modifies an existing variable found by walking up the parent environments.

```
x <- 0
f <- function() {
g <- function() {
x <<- 2
}
x <- 1
g()
x
}
f()
#> [1] 2
x
#> [1] 0
h <- function() {
x <- 1
x <<- 2
x
}
```

```
h()
#> [1] 1
x
#> [1] 2
```

- ▷ If <<- doesn't find an existing variable, it will create one in the global environment. This is usually undesirable, because global variables introduce non-obvious dependencies between functions.
- ▷ name <<- value is equivalent to assign("name", value, inherits = TRUE).

```
rebind <- function(name, value, env = parent.frame()) {
if (identical(env, emptyenv())) {
stop("Can't find ", name, call. = FALSE)
}
if (exists(name, envir = env, inherits = FALSE)) {
assign(name, value, envir = env)
}
else {
rebind(name, value, parent.env(env))
}
}
rebind("a", 10)
a <- 5
rebind("a", 10)
a
#> [1] 10
f <- function() {
g <- function() {
rebind("x", 2)
}
x <- 1
g()
x
}
f()
#> [1] 2
```

◇ **Delayed bindings**

- ▷ Rather than assigning the result of an expression immediately, it creates and stores a promise to evaluate the expression when needed (much like the default lazy evaluation of arguments in R functions)
- ▷ We can create delayed bindings with the special assignment operator %<d-%, provided by the pryr package.[22]

```
library(pryr) a %<d-% 1
system.time(a)
#> user system elapsed
#> 0 0 0
```

[22]Note that we need to be careful with more complicated expressions because user-created infix functions have very high precedence. They're higher in precedence than every other infix operator apart from ^, $, @, and ::.

- ▷ %<d-% is a wrapper around the base delayedAssign() function, which you may need to use directly if you need more control. delayedAssign() has four parameters:
  - ◇ x: a variable name given as a quoted string
  - ◇ value: an unquoted expression to be assigned to x
  - ◇ eval.env: the environment in which to evaluate the expression
  - ◇ assign.env: the environment in which to create the binding
- ▷ One application of delayedAssign is autoload, a function that powers library(). autoload makes R behave as if the code and data in a package is loaded in memory, but it doesn't actually do any work until you call one of the functions or access a dataset.

◇ **Active bindings**

- ▷ You can create active bindings where the value is recomputed every time you access the name:

```
x %<a-% runif(1)
x
#> [1] 0.4428
x
#> [1] 0.3233
```

- ▷ %<a-% is a wrapper for the base function makeActiveBinding(). You may want to use this function directly if you want more control. It has three arguments:
  - ◇ sym: a variable name, represented as a name object or a string
  - ◇ fun: a single argument function. Getting the value of sym calls fun with zero arguments, and setting the value of sym calls fun with one argument, the value.
  - ◇ env: the environment in which to create the binding.

2.7 Exceptions and debugging

Debugging, condition handling and defensive programming

◇ **Debugging**: how to fix unanticipated problems

◇ **Condition handling**: how functions can communicate problems and how you can take action based on those communications.[23]

- ▷ Fatal errors are raised by stop() and force all execution to terminate. Errors are used when there is no way for a function to continue.
- ▷ Warnings are generated by warning() and are used to display potential problems, such as when some elements of a vectorised input are invalid, like log(-1:2).
- ▷ Messages are generated by message() and are used to give informative output in a way that can easily be suppressed by the user (?suppressMessages()).

Recommended readings for condition handling in R:

1. A prototype of a condition system for R (by Robert Gentleman & Luke Tierney) http://homepage.stat.uiowa.edu/~luke/R/exceptions/simpcond.html
   This describes an early version of R's condition system. The implementation has changed somewhat since this was written, but it provides a good overview of how the pieces fit together, and some motivation for the design.

[23]Function authors can also communicate with their users with print() or cat(), but I think that's a bad idea because it's hard to capture and selectively ignore this sort of output. Printed output is not a condition, so you can't use any of the useful condition handling tools you'll learn about below.

2. Beyond Exception Handling: Conditions and Restarts (by Peter Seibel) `http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html` or `http://adv-r.had.co.nz/beyond-exception-handling.html`

   This describes exception handling in Lisp, which happens to be very similar to R's approach. It provides useful motivation and more sophisticated examples.

◇ **Defensive programming**: how to avoid common problems before they occur

- ▷ The basic principle of defensive programming is to *"fail fast"*, to raise an error as soon as you know there's something wrong, rather than trying to silently struggle through.
- ▷ In R, this has three particular applications: checking that inputs are correct, avoiding non-standard evaluation, and avoiding functions that can return different types of output.

Debugging techniques [24]

There are four steps:

1. **Realize that you have a bug**

   You can't fix a bug until you know it exists. This is one reason why automated test suites are important when producing high-quality code. Read more about automated testing here: `http://adv-r.had.co.nz/Testing.html`

2. **Make it repeatable**

   - ◇ Generally, you will start with a big block of code that you know causes the error and then slowly whittle it down to get to the smallest possible snippet that still causes the error.
   - ◇ **Binary search** is particularly useful for this. To do a binary search, you repeatedly remove half of the code until you find the bug. This is fast because, with each step, you reduce the amount of code to look through by half.
   - ◇ If you're using automated testing, this is also a good time to create an automated test case.

3. **Figure out where it is**

   It's a great idea to adopt the scientific method. Generate hypotheses, design experiments to test them and record your results.

4. **Fix it and test it**

   It's very useful to have automated tests in place. Not only does this help to ensure that you've actually fixed the bug, it also helps to ensure you haven't introduced any new bugs in the process.

   In the absence of automated tests, make sure to carefully record the correct output, and check against the inputs that previously failed.

Debugging tools

◇ There are three key debugging tools:[25]

- ▷ RStudio's error inspector and `traceback()` which list the sequence of calls that lead to the error.

[24] Finding your bug is a process of confirming the many things that you believe are true — until you find one which is not true. — Norm Matloff

[25] You shouldn't need to use these tools when writing new functions.Instead of trying to write one big function all at once, work interactively on small pieces. If you start small, you can quickly identify why something doesn't work. But if you start large, you may end up struggling to identify the source of the problem.

- RStudio's "Rerun with Debug" tool and options(error = browser) which open an interactive session where the error occurred.
- RStudio's breakpoints and browser() which open an interactive session at an arbitrary location in the code.

♢ **Determing the sequence of calls**

- The first tool is the **call stack**, the sequence of calls that lead up to an error. traceback()
- If you're calling code that you source()d into R, the traceback will also display the location of the function, in the form filename.r#linenumber.[26]
- Sometimes this is enough information to let you track down the error and fix it. However, it's usually not: it shows you where the error occurred, but not why.

♢ **Browsing on error**

- You can use the error option which specifies a function to run when an error occurs. The function most similar to Rstudio's debug is browser(): this will start an interactive console in the environment where the error occurred.
- Use options(error = browser) to turn it on, re-run the previous command, then use options(error = NULL) to return to the default error behaviour. We can also automate this with the following function:

```
browseOnce <- function() {
old <- getOption("error")
function() {
options(error = old)
browser()
}
}
options(error = browseOnce())
f <- function() stop("!")
# Enters browser
f()
# Runs normally
f()
```

- There are two other useful functions that you can use with the error option:
  - recover is a step up from browser, as it allows you to enter the environment of any of the calls in the call stack.[27]
  - dump.frames is an equivalent to recover for non-interactive code. It creates a last.dump.rda file in the current working directory. Then, in a later interactive R session, you load that file, and use debugger() to enter an interactive debugger with the same interface as recover(). This allows interactive debugging of batch code.

```
# In batch R process —-
dump_and_quit <- function() {
# Save debugging info to file last.dump.rda
dump.frames(to.file = TRUE)
```

[26]These are clickable in Rstudio, and will take you to the corresponding line of code in the editor.
[27]This is useful because often the root cause of the error is a number of calls back.

```
# Quit R with error status
q(status = 1)
}
options(error = dump_and_quit)
# In a later interactive session —-
load("last.dump.rda")
debugger()
```

◇ To reset error behaviour to the default, use options(error = NULL). Then errors will print a message and abort function execution.

♢ Browsing arbitrary code

- ▷ As well as entering an interactive console on error, you can enter it at an arbitrary code location by using either an Rstudio breakpoint or browser().
- ▷ There are two small downsides to breakpoints:
  - ◇ There are few unusual situations in which breakpoints will not work: read breakpoint troubleshooting (http://www.rstudio.com/ide/docs/debugging/breakpoint-troubleshooting)for more details.
  - ◇ Rstudio currently does not support conditional breakpoints, whereas you can always put browser() inside an if statement.
- ▷ As well as adding browser() yourself, there are two functions that will add it to code for you:
  - ◇ debug() inserts a browser statement in the first line of the specified function. undebug() will remove it, or you can use debugonce() to browse only on the next run.
  - ◇ utils::setBreakpoint() works similarly, but instead of taking a function name, it takes a file name and line number and finds the appropriate function for you.
- ▷ These two functions are both special cases of trace(), which inserts arbitrary code at any position in an existing function.
- ▷ trace() is occasionally useful when you're debugging code that you don't have the source for. To remove tracing from a function, use untrace().
- ▷ You can only perform one trace per function, but that one trace can call multiple functions.

♢ The call stack: traceback(), where and recover().

- ▷ Unfortunately the call stacks printed by traceback(), browser() + where and recover() are not consistent.
- ▷ The following table shows how the call stacks from a simple nested set of calls are displayed by the three tools

traceback()	where	recover()
4: stop("Error")	where 1: stop("Error")	1: f()
3: h(x)	where 2: h(x)	2: g(x)
2: g(x)	where 3: g(x)	3: h(x)
1: f()	where 4: f()	

  - ◇ Note that numbering is different between traceback() and where, and recover() displays calls in the opposite order, and omits the call to stop().

♢ Other types of failure

- ▷ A function may generate an unexpected warning.

- ◇ The easiest way to track down warnings is to convert them into errors with `options(warn = 2)` and use the regular debugging tools.
- ◇ When you do this you'll see some extra calls in the call stack, like `doWithOneRestart()`, `withOneRestart()`, `withRestarts()` and `.signalSimpleWarning()`. Ignore these: they are internal functions used to turn warnings into errors.

▷ A function may generate an unexpected message.

- ◇ There's no built in tool to help solve this problem, but it's possible to create one:[28]

```
message2error <- function(code) {
withCallingHandlers(code, message = function(e) stop(e))
}
f <- function() g()
g <- function() message("Hi!")
g()
message2error(g())
traceback()
```

▷ A function might never return.

This is particularly hard to debug automatically, but sometimes terminating the function and looking at the call stack is informative. Otherwise, use the basic debugging strategies described above.

▷ The worst scenario is that your code might crash R completely, leaving you with no way to interactively debug your code.

- ◇ This indicates a bug in underlying C code and is hard to debug. Sometimes an interactive debugger, like gdb, can be useful, but describing how to use it is beyond the scope of this book.

Condition handlng

Some errors, however, are expected, and you want to handle them automatically.
In R, expected errors crop up most frequently when you're fitting many models to different datasets, such as bootstrap replicates.
Sometimes the model might fail to fit and throw an error, but you don't want to stop everything; instead you want to fit as many models as possible and then perform diagnostics after the fact.
In R, there are three tools for handling conditions (including errors) programmatically:

1. **Ignore errors with try()**

   - ◇ gives you the ability to continue execution even when an error occurs.
   - ◇ If you wrap the statement that creates the error in `try()`, the error message will be printed but execution will continue.
   - ◇ You can suppress the message with `try(..., silent = TRUE)`. To pass larger blocks of code to `try()`, wrap them in {}:
   - ◇ You can also capture the output of the `try()` function. If successful, it will be the last result evaluated in the block (just like a function); if unsuccessful it will be an (invisible) object of class "try-error":

   ```
   success <- try(1 + 2)
   failure <- try("a" + "b")
   ```

[28]As with warnings, you'll need to ignore some of the calls on the tracback (i.e. the first two and the last seven).

```
str(success)
#> num 3
str(failure)
#> Class 'try-error' atomic [1:1] Error in "a" + "b" : non-numeric argument to binary operator
#>
#> ..- attr(*, "condition")=List of 2
#> .. ..$ message: chr "non-numeric argument to binary operator"
#> .. ..$ call : language "a" + "b"
#> .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

- try() is particularly useful when you're applying a function to multiple elements in a list:

```
elements <- list(1:10, c(-1, 10), c(T, F), letters)
results <- lapply(elements, log)
#> Warning: NaNs produced
#> Error: non-numeric argument to mathematical function
results <- lapply(elements, function(x) try(log(x)))
#> Warning: NaNs produced
```

- Then you can easily find the locations of errors with sapply(), and extract the successes or look at the inputs that lead to failures.

```
is.error <- function(x) inherits(x, "try-error")
succeeded <- !sapply(results, is.error)
# look at successful results str(results[succeeded])
#> List of 3
#> $ : num [1:10] 0 0.693 1.099 1.386 1.609 ...
#> $ : num [1:2] NaN 2.3
#> $ : num [1:2] 0 -Inf
# look at inputs that failed str(elements[!succeeded])
#> List of 1
#> $ : chr [1:26] "a" "b" "c" "d" ...
```

- Another useful try() idiom is using a default value if an expression fails. Simply assign the default value outside the try block, and then run the risky code:

```
default <- NULL
try(default <- read.csv("possibly-bad-input.csv"), silent = TRUE)
```

2. **Handle conditions with tryCatch()**

   - lets you specify **handler** functions that control what happens when a condition is signalled.
   - tryCatch() is a general tool for handling conditions: as well as errors you can take different actions for warnings, messages and interrupts.[29]
   - With tryCatch() you map conditions to handlers, named functions that are passed the condition as an input. If a condition is signalled, tryCatch will call the first handler whose name matches one of the classes of the condition.
   - The only useful built-in names are error, warning, message, interrupt and the catch-all condition.

[29]They can't be generated directly by the programmer, but are raised when the user attempts to terminate execution by pressing Ctrl + Break, Escape, or Ctrl + C (depending on the platform).

- A handler function can do anything, but typically it will either return a value or create a more informative error message:

```
show_condition <- function(code) {
tryCatch(code,
error = function(c) "error",
warning = function(c) "warning",
message = function(c) "message"
)
}
show_condition(stop("!"))
#> [1] "error" show_condition(warning("?!"))
#> [1] "warning" show_condition(message("?"))
#> [1] "message"
# If no condition is captured, tryCatch returns the value of the input
show_condition(10)
#> [1] 10
```

- You can use tryCatch() to implement try().[30]

```
try2 <- function(code, silent = FALSE) {
tryCatch(code, error = function(c) {
msg <- conditionMessage(c)
if (!silent) message("Error: ", c)
invisible(structure(msg, class = "try-error"))
})
}
try2(1)
#> [1] 1
try2(stop("Hi"))
#> Error: Error in doTryCatch(return(expr), name, parentenv, handler): Hi
try2(stop("Hi"), silent = TRUE)
```

- As well as returning default values when a condition is signalled, handlers can be used to make more informative error messages.

```
read.csv2 <- function(file, ...) {
tryCatch(read.csv(file, ...), error = function(c) {
c$message <- paste0(c$message, " ( in ", file, ")")
stop(c)
})
}
read.csv("code/dummy.csv")
#> Error: duplicate 'row.names' are not allowed
read.csv2("code/dummy.csv")
#> Error: duplicate 'row.names' are not allowed ( in code/dummy.csv)
```

[30]Note the use of conditionMessage() to extract the message associated with the original error.

- Catching interrupts can be useful if you want to take special action when the user tries to abort running code.[31]

  ```
  # Don't let the user interrupt the code
  i <- 1
  while(i < 3) {
  tryCatch({
  Sys.sleep(0.5)
  message("Try to escape")
  }, interrupt = function(x) {
  message("Try again!") i <<- i + 1
  })
  }
  ```

- tryCatch() has one other argument: finally, which specifies a block of code (not a function) to run regardless of whether of the initial expression succeeds or fails.

3. **withCallingHandlers():**

   - is a variant of tryCatch() that runs its handlers in a different context. It is rarely needed, but is useful to be aware of.
   - There are two main differences between the functions:
     - The return value of tryCatch() handlers is returned by tryCatch(), where the return value of withCallingHandlers() handlers is ignored:

       ```
       f <- function() stop("!")
       tryCatch(f(), error = function(e) 1)
       #> [1] 1
       withCallingHandlers(f(), error = function(e) 1)
       #> Error: !
       ```

     - The handlers in withCallingHandlers() are called in the context of the call that generated the condition; the handlers in tryCatch() are called in the context of tryCatch().

       ```
       f <- function()
       g() g <- function()
       h() h <- function() stop("!")
       tryCatch(f(), error = function(e) print(sys.calls())) [32]
       # [[1]] tryCatch(f(), error = function(e) print(sys.calls()))
       # [[2]] tryCatchList(expr, classes, parentenv, handlers)
       # [[3]] tryCatchOne(expr, names, parentenv, handlers[[1L]])
       # [[4]] value[[3L]](cond)
       withCallingHandlers(f(), error = function(e) print(sys.calls()))
       # [[1]] withCallingHandlers(f(), error = function(e) print(sys.calls()))
       # [[2]] f()
       # [[3]] g()
       # [[4]] h()
       # [[5]] stop("!")
       # [[6]] .handleSimpleError(function (e) print(sys.calls()), "!", quote(h()))
       # [[7]] h(simpleError(msg, call))
       ```

[31] This can be useful for clean up (e.g. deleting files, closing connections). This is functionally equivalent to using on.exit() but it can wrap smaller chunks of code than an entire function.

[32] sys.calls() is the run-time equivalent of traceback(), listing all calls leading to the current function.

◇ These subtle differences are rarely useful, except when you're trying to capture exactly what went wrong and pass it on to another function. For most purposes, you should never need to use withCallingHandlers().

◇ **Custom signal classes**

- ▷ One of the challenges of error handling in R is that most functions just call stop() with a string.
- ▷ This is error prone, not only because the text of the error might change over time, but also because many error messages are translated, so the message might be completely different to what you expect.
- ▷ Conditions are S3 classes, so you can define your own classes if you want to distinguish different types of error.
- ▷ Each condition signalling function, stop(), warning() and message() can be given either a list of strings, or a custom S3 condition object.
- ▷ Custom condition objects are not used very often, but are very useful because they make it possible for the user to respond to different errors in different ways.
- ▷ Custom constructor function for conditions: must contain message and call components, and may contain other useful components; when creating a new condition, it should always inherit from conditon and one of error, warning and message:

```
condition <- function(subclass, message, call = sys.call(-1), ...) {
structure(
class = c(subclass, "condition"),
list(message = message, call = call),
...
)
}
is.condition <- function(x) inherits(x, "condition")
```

- ▷ You can signal an arbitrary condition with signalCondition(), but nothing will happen unless you've instantiated a custom signal handler (with tryCatch() or withCallingHandlers().
- ▷ Instead, use stop(), warning() or message() as appropriate to trigger the usual handling. R won't complain if the class of your condition doesn't match the function, but you should avoid this in real code.
- ▷ You can then use tryCatch() to take different actions for different types of errors.

```
custom_stop <- function(subclass, message, call = sys.call(-1), ...) {
c <- condition(c(subclass, "error"), message, call = call, ...)
stop(c)
}
my_log <- function(x) {
if (!is.numeric(x))
custom_stop("invalid_class", "my_log() needs numeric input")
if (any(x < 0))
custom_stop("invalid_value", "my_log() needs positive inputs")
log(x)
}
tryCatch(
my_log("a"),
invalid_class = function(c) "class",
invalid_value = function(c) "value"
)
#> [1] "class"
```

▷ Note that when using **tryCatch()** with multiple handlers and custom classes, the first handler to match any class in the signal's class hierarchy is called, not the best match. For this reason, you need to make sure to put the most specific handlers first:

```
tryCatch(customStop("my_error", "!"),
error = function(c) "error",
my_error = function(c) "my_error"
)
#> [1] "error"
tryCatch(custom_stop("my_error", "!"),
my_error = function(c) "my_error",
error = function(c) "error" )
#> [1] "my_error"
```

Defensive programming

◇ Defensive programming is the art of making code fail in a well-defined manner even when something unexpected occurs.

◇ A key principle of defensive programming is to "fail fast": as soon as you discover something is wrong, signal an error.

◇ The principle of "fail fast" has three main applications in R:

- ▷ Be strict about what you accept.[33]
- ▷ Avoid functions that use non-standard evaluation, like subset, transform, and with.
  These functions save time when used interactively, but because they make assumptions to reduce typing, when they fail, they often fail with uninformative error messages.
- ▷ Avoid functions that return different types of output depending on their input.
  The two biggest offenders are [and sapply(). Whenever subsetting a data frame in a function, you should always use drop = FALSE, otherwise you will accidentally convert 1-column data frames into vectors. Similarly, never use sapply() inside a function: always use the stricter vapply() which will throw an error if the inputs are incorrect types and return the correct type of output even for zero-length inputs.
- ▷ There is a tension between interactive analysis and programming.
  - ◇ When you're doing an analysis, you want R to do what you mean, and if it guesses wrong, you'll discover it right away and you can fix it.
  - ◇ When you're programming, you want functions with no magic that signal errors is anything is slightly wrong or underspecified.
  - ◇ Keep this tension in mind when writing functions: If you're making a function to facilitate interactive data analysis, feel free to guess what the analyst wants and recover from minor misspecifications automatically; if you're making a function to program with, be strict, and never make guesses about what the caller wants.

[33]You can use stopifnot(), the assertthat package or simple if statements and stop().

3 Functional Programming

3.1 Functional programming

- At its heart, R is a **functional programming** (FP) language; it focusses on the creation and manipulation of functions.
- R has what's known as first class functions, functions that can be[34]:
  - created without a name,
  - assigned to variables and stored in lists,
  - returned from functions,
  - and passed as arguments to other functions.

Motivation

- Repetition is bad because it allows for inconsistencies (aka bugs), and it makes the code harder to change.
- The "do not repeat yourself", or DRY, principle, was popularised by the pragmatic programmers, Dave Thomas and Andy Hunt. This principle states that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system".
- The ideas of FP are important because they give us new tools to reduce duplication.
- lapply() is called a functional, because it takes a function as an argument. We can use lapply() with one small trick: rather than simply assigning the results to df we assign them to df[], so R's usual subsetting rules take over and we get a data frame instead of a list.

```
fix_missing <- function(x) {
x[x == -99] <- NA
x }
df[] <- lapply(df, fix_missing)
```

- Writing simple functions that can be understood in isolation and then composed together to solve complex problems is an important technique for effective FP.
- Take advantage of another functional programming technique, storing functions in lists, to remove this duplication:

```
summary <- function(x) {
funs <- c(mean, median, sd, mad, IQR)
lapply(funs, function(f) f(x, na.rm = TRUE))
}
```

[34] This means that you can do anything with functions that you can do with vectors: you can create them inside other functions, pass them as arguments to functions, return them as results from functions and store multiple functions in a list.

Anonymous functions

- ♢ In R, functions are objects in their own right. They aren't automatically bound to a name and R doesn't have a special syntax for creating named functions, unlike C, C++, Python or Ruby.
- ♢ Given the name of a function, like "mean", it's possible to find the function using match.fun(). You can't do the opposite: given the object f <- mean, there's no way to find its name.
- ♢ Not all functions have a name, and some functions have more than one name. Functions that don't have a name are called **anonymous functions**.
- ♢ Like all functions in R, anoynmous functions have formals(), a body(), and a parent environment():

```
formals(function(x = 4) g(x) + h(x))
#> $x
#> [1] 4
body(function(x = 4) g(x) + h(x))
#> g(x) + h(x)
environment(function(x = 4) g(x) + h(x))
#> <environment: 0x1ab1220>
```

- ♢ You can call anonymous functions directly, but the code is a little tricky to read because you must use parentheses in two different ways: first, to call a function, and second to make it clear that you want to call the anonymous function function(x) 3, as opposed to calling a function named '3' (which isn't a valid function name) inside the anonymous function:[35]

```
(function(x) x + 3)(10)
#> [1] 13
# Exactly the same as
f <- function(x) x + 3 f(10)
#> [1] 13
# Doesn't do what you expect
function(x) 3()
#> function(x) 3()
#> <environment: 0x1ab1220>
```

- ♢ One of the most common uses for anonymous functions is to create closures, functions made by other functions.
- ♢ A good rule of thumb is that an anonymous function should fit on one line and shouldn't need to use {}.

[35]You can supply arguments to anonymous functions in all the usual ways (by position, exact name and partial name) but if you find yourself doing this, it's probably a sign that your function needs a name.

Introduction to closures[36]

- One use of anonymous functions is to create small functions that it's not worth naming; the other main use of anonymous functions is to create closures, functions written by functions.
- Closures are so called because they **enclose** the environment of the parent function, and can access all variables in the parent.
- This is useful because it allows us to have two levels of parameters. One level of parameters (the parent) controls how the function works; the other level (the child) does the work.
- In R, almost every function is a closure, because all functions remember the environment in which they are created, typically either the global environment, if it's a function that you've written, or a package environment, if it's a function that someone else has written.
- The only exception are primitive functions, which call to C directly.
- One way to see the contents of the environment is to convert it to a list:

```
power <- function(exponent) { function(x) x ^ exponent }
square <- power(2)
as.list(environment(square))
#> $exponent
#> [1] 2
```

- Another way to see what's going on is to use pryr::unenclose(), which substitutes the variables defined in the enclosing environment into the original functon:

```
library(pryr)
unenclose(square)
#> function (x)
#> x^2
#> <environment: 0x1ab1220>
```

- Note that the parent environment of the closure is the environment created when the parent function is **called.**
- This environment normally disappears once the function finishes executing, but because we return a function, the environment is captured and attached to the new function.
- Each time we re-run power() a new environment is created, so each function produced by power is independent.
- **Function factories**
  - Function factories are most useful when:
    - the different levels are more complex, with multiple arguments and complicated bodies
    - some work only needs to be done once, when the function is generated
- **Mutable state**

[36] "An object is data with functions. A closure is a function with data." — John D Cook

- ▷ Having variables at two levels makes it possible to maintain state across function invocations, because while the function environment is refreshed every time, its parent environment stays constant.
- ▷ The key to managing variables at different levels is the double arrow assignment operator (<<-).
- ▷ Unlike the usual single arrow assignment (<-) that always assigns in the current environment, the double arrow operator will keep looking up the chain of parent environments until it finds a matching name.[37][38]

```
new_counter <- function() {
i <- 0
function() {
i <<- i + 1
i }
}
```

The new function is a closure, and its enclosing environment is the usually temporary environment created when new_counter is run. When the closures counter_one and counter_two are run, each one modifies the counter in a different enclosing environment and so maintain different counts.

```
counter_one <- new_counter()
counter_two <- new_counter()
counter_one() # -> [1] 1
#> [1] 1
counter_one() # -> [1] 2
#> [1] 2
counter_two() # -> [1] 1
#> [1] 1
```

- ♢ The counters get around the "fresh start" limitation by not modifying variables in their local environment. Since the changes are made in the unchanging parent (or enclosing) environment, they are preserved across function calls.
- ♢ Modifying values in a parent environment is an important technique because it is one way to generate "mutable state" in R.
- ♢ Mutable state is normally hard to achieve, because every time it looks like you're modifying an object, you're actually creating a copy and modifying that.
- ♢ That said, if you do need mutable objects, except in the simplest of cases, it's usually better to use the RC OO system. RC objects are easier to document, and provide easier ways to inherit behaviour.

Lists of functions

- ♢ In R, functions can be stored in lists. Instead of giving a set of functions related names, you can store them in a list.
- ♢ This makes it easier to work with groups of related functions, in the same way a data frame makes it easier to work with groups of related vectors.

[37] Together, a static parent environment and <<- make it possible to maintain state across function calls.

[38] <- will not modify the value in the parent environment (only the current environment, the one created when calling the function), while <<- will do.

◇ If we want to call each functions to check that we've implemented them correctly and they return the same answer, we can use lapply(), either with an anonymous function, or an equivalent named function.

```
lapply(compute_mean, function(f, ...) f(...), x)
```

or

```
call_fun <- function(f, ...) f(...)
lapply(compute_mean, call_fun, x)
```

◇ If we would like to remove the missings we can do:

```
lapply(funs, function(f) f(x, na.rm = TRUE))
```

or

```
# Or use a named function instead of an anonymous function
remove_missings <- function(f) {
function(...) f(..., na.rm = TRUE)
}
funs2 <- lapply(funs, remove_missings)
```

◇ **Moving lists of functions to the global environment**

- ▷ Example:

  ```
  simple_tag <- function(tag) {
  function(...) paste0("<", tag, ">", paste0(...), "</", tag, ">")
  }
  html <- list(
  p = simple_tag("p"),
  b = simple_tag("b"),
  i = simple_tag("i"),
  img = function(path, width, height) {
  paste0("<img src='", path, "' width='", width, "' height = '", height, '" />')
  }
  )
  ```

- ▷ We store the functions in a list because we don't want them to be available all the time: the risk of a conflict between an existing R function and an HTML tag is high. However, keeping them in a list means that our code is more verbose than necessary: we need to use $ every time if we would like to use the function under the list.
- ▷ We have three options ot eliminate the use of html$:[39]
  1. For a very temporary effect, we can use a **with()** block:
     ```
     with(html, p("This is ", b("bold"), ", ", i("italic"), " and ", b(i("bold italic")), " text"))
     ```
  2. For a longer effect, we can use attach() to add the functions in html in to the search path. It's posible to undo this action using detach:
     ```
     attach(html)
     p("This is ", b("bold"), ", ", i("italic"), " and ", b(i("bold italic")), " text")
     #> [1] "<p>This is <b>bold</b>, <i>italic</i> and <b><i>bold italic</i></b> text</p>"

     detach(html)
     ```

[39]The author recommends the first option because it makes it very clear what's going on, and when code is being executed in a special context.

3. Finally, we could copy the functions into the global environment with list2env().

```
list2env(html, environment())
#> <environment: 0x1ab1220>
p("This is ", b("bold"), ", ", i("italic"), " and ", b(i("bold italic")), " text")
#> [1] "<p>This is <b>bold</b>, <i>italic</i> and <b><i>bold italic</i></b> text</p>"

rm(list = names(html), envir = environment())
```

Case study: numerical integration

- ♢ The idea behind numerical integration is simple: we want to find the area under the curve by approximating a complex curve with simpler components.
- ♢ The two simpliest approaches are the midpoint and trapezoid rules; the mid point rule approximates a curve by a rectangle, and the trapezoid rule by a trapezoid.
- ♢ break up the range into smaller pieces and integrate each piece using one of the simple rules. This is called **composite integration.**

```
midpoint_composite <- function(f, a, b, n = 10) {
points <- seq(a, b, length = n + 1)
h <- (b - a) / n
area <- 0
for (i in seq_len(n)) {
area <- area + h * f((points[i] + points[i + 1]) / 2)
}
area
}
trapezoid_composite <- function(f, a, b, n = 10) {
points <- seq(a, b, length = n + 1)
h <- (b - a) / n
area <- 0
for (i in seq_len(n)) {
area <- area + h / 2 * (f(points[i]) + f(points[i + 1]))
}
area
}
composite <- function(f, a, b, n = 10, rule) {
points <- seq(a, b, length = n + 1)
area <- 0
for (i in seq_len(n)) {
area <- area + rule(f, points[i], points[i + 1]) }
area
}
```

```
composite(sin, 0, pi, n = 10, rule = midpoint)
#> [1] 2.008
composite(sin, 0, pi, n = 10, rule = trapezoid)
#> [1] 1.984
```

◇ It turns out that the midpoint, trapezoid, Simpson and Boole rules are all examples of a more general family called **Newton-Cotes** rules. (They are polynomials of increasing complexity).

```
# From http://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas
newton_cotes <- function(coef, open = FALSE) {
n <- length(coef) + open
function(f, a, b) {
pos <- function(i) a + i * (b - a) / n
points <- pos(seq.int(0, length(coef) - 1))
(b - a) / sum(coef) * sum(f(points) * coef)
}
}
rules <- list(
trapezoid = newton_cotes(c(1, 1)),
midpoint = newton_cotes(1, open = TRUE),
simpson = newton_cotes(c(1, 4, 1)),
boole = newton_cotes(c(7, 32, 12, 32, 7)),
milne = newton_cotes(c(2, -1, 2), open = TRUE)
)
```

◇ Mathematically, the next step in improving numerical integration is to move from a grid of evenly spaced points to a grid where the points are closer together near the end of the range, such as **Gaussian quadrature**.

3.2 Functionals[40]

Introduction

◇ Higher-order functions encompass any functions that either take a function as an input or return a function as output.

◇ The complement to a closure is a **functional**, a function that takes a function as an input and returns a vector as output.

◇ Since functions are first class objects in R, there's no difference between calling a function with a vector or function as input.

◇ Many functionals (like lapply()) offer alternatives to for loops. For loops have a bad rap in R, and some programmers try to eliminate them at all costs.

[40]"To become significantly more reliable, code must become more transparent. In particular, nested conditions and loops must be viewed with great suspicion. Complicated control flows confuse programmers. Messy code often hides bugs." — Bjarne Stroustrup

◇ The real downside of for loops is that they're not very expressive. A for loop conveys that you're iterating over something, but it doesn't communicate the higher-level task you're trying to complete.

◇ Functionals are not as general as for loops, but by being more specific they allow you to communicate more clearly. A functional allows you to say I want to transform each element of this list, or each row of this array.

◇ As well as more clearly communicating intent, functionals reduce the chances of bugs, and can be more efficient.

◇ As well as replacements for for loops, functionals do play other roles. They are also useful tools for encapsulating common data manipulation tasks, the split-apply-combine pattern; for thinking "functionally"; and for working with mathematical functions.

◇ This will not always produce the fastest code, but it is a mistake to focus on speed until you know it will be a problem. Once you do have clear, correct code you can make it fast using the techniques in performance.

My first functional: lapply()

◇ lapply() is written in C for performance, but we can create a simple R implementation that works the same way[41]:

```
lapply2 <- function(x, f, ...) {
out <- vector("list", length(x))
for (i in seq_along(x)) {
out[[i]] <- f(x[[i]], ...)
}
out
}
```

◇ From this code, you can see that lapply() is a wrapper around a common for loop pattern: we create a space for output, and then fill it in, applying f() to each component of the list.

◇ All other for loop functionals build on this base, modifying either the input, the output, or what data the function is applied to.

◇ Since data frames are also lists, lapply() is useful when you want to do something to each column of a data frame.

◇ The pieces of x are always supplied as the first argument to f. You can override this using R's regular function calling semantics, supplying additional named arguments. So to use lapply() with the second argument, we just need to name the first argument:

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(100)
unlist(lapply(trims, mean, x = x))
#> [1] 12.16175 -0.09112 -0.08860 0.01817
```

◇ **Looping patterns**

[41] From this code you can see that lapply() will also works with vectors: both length() and '[[work the same way for lists and vectors.

- ▷ When using lapply() and friends, it's useful to remember that there are usually three ways to loop over an vector:
  1. loop over the elements of the vector: for(x in xs)
  2. loop over the numeric indices of the vector: for(i in seq_along(xs))
  3. loop over the names of the vector: for(nm in names(xs))
- ▷ If you're saving the results from a for loop, you usually can't use the first form because it makes very inefficient code. When extending an existing data structure, all the existing data must be copied every time you extend it.
- ▷ It's much better to create enough space for the output and then fill it in, using the second looping form.
- ▷ Corresponding to the three ways to use a for loop there are three ways to use lapply() with an object:[42]
  lapply(xs, function(x) {})
  lapply(seq_along(xs), function(i) {})
  lapply(names(xs), function(nm) {})

For loop functional: friends of lapply()

- ♢ The art of using functionals is to recognise what common looping patterns are implemented in existing base functionals, and then use them instead of loops.
- ♢ if you discover you're duplicating the same looping pattern in many places, you should extract it out into its own function.
- ♢ **Vector output: sapply and vapply**
  - ▷ sapply() and vapply() are very similar to lapply() except they will simplify their output to produce an atomic vector.
  - ▷ sapply() guesses, while vapply() takes an additional argument specifying the output type.
  - ▷ sapply() is useful for interactive use because it saves typing, but if you use it inside your functions you will get weird errors if you supply the wrong type of input.
  - ▷ vapply() is more verbose, but gives more informative error messages and never fails silently, so is better suited for use inside other functions.
  - ▷ When given a data frame sapply() and vapply() give the same results.
  - ▷ When given an empty list, sapply() has no basis to guess the correct type of output, and returns NULL, instead of the more correct zero-length logical vector.
  - ▷ If the function returns results of different types or lengths, sapply() will silently return a list, while vapply() will throw an error.[43]
  - ▷ sapply() is a thin wrapper around lapply(), transforming a list into a vector in the final step; vapply() reimplements lapply() but assigns results into a vector (or matrix) of the appropriate type instead of into a list.
- ♢ **Multiple inputs: Map (and mapply)**

[42] Typically you use the first form because lapply() takes care of saving the output for you. However, if you need to know the position or the name of the element you're working with, you'll need to use the second or third form; they give you both the position of the object (i, nm) and its value (xs[[i]], xs[[nm]]).

[43] sapply() is fine for interactive use because you'll normally notice if something went wrong, but it's dangerous when writing functions.

- With lapply(), only one argument to the function varies; the others are fixed. This makes it poorly suited for some problems.
- Map, a variant of lapply(), where all arguments vary:
  ```
  unlist(Map(weighted.mean, xs, ws))
  ```
  [44]
  ```
  #> [1] 0.6704 0.6874 0.5362 0.5220 0.6120 0.4238 0.3535 0.4999 0.5025 0.4427
  ```
- This is equivalent to
  ```
  stopifnot(length(x) == length(w))
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
  out[[i]] <- weighted.mean(x[[i]], w[[i]])
  }
  ```
- There's a natural equivalence between Map() and lapply() because you can always convert a Map() to an lapply() that iterates over indices, but using Map() is more concise, and more clearly indicates what you're trying to do.
- Map is useful whenever you have two (or more) lists (or data frames) that you need to process in parallel.
- If some of the arguments should be fixed, and not varying, you need to use an anonymous function:
  ```
  Map(function(x, w) weighted.mean(x, w, na.rm = TRUE), xs, ws)
  ```
- Compared to mapply():
  - Map() is equivalent to mapply with simplify = FALSE, which is almost always what you want.
  - Instead of using an anonymous function to provide constant inputs, mapply has the MoreArgs argument which takes a list of extra arguments that will be supplied, as is, to each call. This breaks R's usual lazy evaluation semantics, and is inconsistent with other functions.

◇ Rolling computations

- You can often create your own by recognising common looping structures and implementing your own wrapper.
  ```
  rollmean <- function(x, n) {
  out <- rep(NA, length(x))
  offset <- trunc(n / 2)
  for (i in (offset + 1):(length(x) - n + offset - 1)) {
  out[i] <- mean(x[(i - offset):(i + offset - 1)]) }
  out
  }
  x <- seq(1, 3, length = 1e2) + runif(1e2) plot(x)
  lines(rollmean(x, 5), col = "blue", lwd = 2)
  lines(rollmean(x, 10), col = "red", lwd = 2)
  ```
- But if the noise was more variable (i.e. it had a longer tail) you might worry that your rolling mean was too sensitive to the occasional outlier and instead implement a rolling median.
- To modify rollmean() to rollmedian() all you need to do is replace mean with median inside the loop, but instead of copying and pasting to create a new function, we could extract the idea of computing a rolling summary into its own function:
  ```
  rollapply <- function(x, n, f, ...) {
  ```

[44]Note that the order of arguments is a little different: with Map() the function is the first argument, with lapply() it's the second.

```
out <- rep(NA, length(x))
offset <- trunc(n / 2)
for (i in (offset + 1):(length(x) - n + offset - 1)) {
out[i] <- f(x[(i - offset):(i + offset - 1)], ...)
}
out
}
plot(x)
lines(rollapply(x, 5, median), col = "red", lwd = 2)
```

▷ The internal loop looks pretty similar to a vapply() loop, so we could rewrite the function as:

```
rollapply <- function(x, n, f, ...) {
offset <- trunc(n / 2)
locs <- (offset + 1):(length(x) - n + offset - 1)
vapply(locs, function(i) f(x[(i - offset):(i + offset - 1)], ...), numeric(1))
}
```

♢ **Parallelisation**

▷ One thing that's interesting about the defintions of lapply() is that because each iteration is isolated from all others, the order in which they are computed doesn't matter.

▷ This has a very important consequence: since we can compute each element in any order, it's easy to dispatch the tasks to different cores, and compute in parallel. This is what mclapply() (and mcMap) in the parallel package do:

```
library(parallel)
unlist(mclapply(1:10, sqrt, mc.cores = 4))
#> [1] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000 3.162
```

▷ In this case mclapply() is actually slower than lapply(), because the cost of the individual computations is low, and some additional work is needed to send the computation to the different cores then collect the results together.

▷ In a more realistic example, we see more of an advantage:

```
boot_lm <- function(i, data, formula) {
boot_df <- function(data) data[sample(nrow(data), rep = T), ]
rsquared <- function(mod) summary(mod)$r.square
rsquared(lm(formula, data = boot_df(data)))
}
system.time(lapply(1:500, boot_lm,formula='mpg ~ wt + disp',data=mtcars))
# user system elapsed
# 1.701 0.110 1.798
system.time(mclapply(1:5,boot_lm,mc.cores=2,formula='mpg ~ wt + disp',data=mtcars))
# user system elapsed
# 0.038 0.070 0.051
```

▷ It is rare to get an exactly linear improvement with increasing number of cores, but if your code uses lapply() or Map(), this is an easy way to improve performance.

Data structure functionals

- ♢ another family of functionals works to eliminate loops for common data manipulation tasks.
  - ▷ base functions for working with matrices: apply(), sweep() and outer()
  - ▷ tapply(), which summarises a vector divided into groups by the values of another vector
  - ▷ the plyr package, which generalises the ideas of tapply() to work with inputs of data frames, lists and arrays, and outputs of data frames, lists, arrays and nothing
- ♢ **Matrix and array operations**
  - ▷ apply() is a variant of sapply() that works with matrices and arrays. You can think of it as an operation that summarises a matrix or array, collapsing each row or column to a single number.
  - ▷ It has four arguments:
    - ⋄ X, the matrix or array to summarise
    - ⋄ MARGIN, an integer vector giving the dimensions to summarise over, 1 = rows, 2 = columns, etc
    - ⋄ FUN, a summary function
    - ⋄ ... other arguments passed on to FUN
  - ▷ There are a few caveats to using apply(): it does not have a simplify argument, so you can never be completely sure what type of output you will get.[45]
  - ▷ apply() is also not idempotent in the sense that if the summary function is the identity operator, the output is not always the same as the input[46]:

```
a1 <- apply(a, 1, identity)
identical(a, a1)
#> [1] FALSE
identical(a, t(a1))
#> [1] TRUE
a2 <- apply(a, 2, identity)
identical(a, a2)
#> [1] TRUE
```

  - ▷ sweep() is a function that allows you to "sweep" out the values of a summary statistic.
  - ▷ It is most often useful in conjunction with apply() and it often used to standardise arrays in some way. The following example scales the rows of a matrix so that all values lie between 0 and 1.

```
x <- matrix(rnorm(20, 0, 10), nrow = 4)
x1 <- sweep(x, 1, apply(x, 1, min))
x2 <- sweep(x1, 1, apply(x1, 1, max), "/")
```

  - ▷ outer() takes multiple vector inputs and creates a matrix or array output where the input function is run over every combination of the inputs:

```
# Create a times table
outer(1:9, 1:9, "*")
#> [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
#> [1,] 1 2 3 4 5 6 7 8 9
#> [2,] 2 4 6 8 10 12 14 16 18
```

[45] This generally means that apply() is not safe to use inside a function, unless you carefully check the inputs.
[46] You can put high-dimensional arrays back in the right order using aperm(), or use plyr::aaply(), which is idempotent.

```
#> [3,] 3 6 9 12 15 18 21 24 27
#> [4,] 4 8 12 16 20 24 28 32 36
#> [5,] 5 10 15 20 25 30 35 40 45
#> [6,] 6 12 18 24 30 36 42 48 54
#> [7,] 7 14 21 28 35 42 49 56 63
#> [8,] 8 16 24 32 40 48 56 64 72
#> [9,] 9 18 27 36 45 54 63 72 81
```

- Good places to learn more about apply() and friends are:
  - "Using apply, sapply, lapply in R" by Peter Werner. http://petewerner.blogspot.com/2012/12/using-apply-sapply-lapply-in-r.html
  - "The infamous apply function" by Slawa Rokicki. http://rforpublichealth.blogspot.no/2012/09/the-infamous-apply-function.html
  - "The R apply function – a tutorial with examples" by axiomOfChoice. http://forgetfulfunctor.blogspot.com/2011/07/r-apply-function-tutorial-with-examples.html
  - The stack overflow question "R Grouping functions: sapply vs. lapply vs. apply. vs. tapply vs. by vs. aggregate vs".http://stackoverflow.com/questions/3505701

Group apply

- You can think about tapply() as a generalisation to apply() that allows for "ragged" arrays, where each row can have different numbers of columns.
- It's easiest to understand how tapply() works by first creating a "ragged" data structure from the inputs. This is the job of the split() function, which takes two inputs and returns a list, where all the elements in the first vector with equal entries in the second vector get put in the same element of the list:
- Then you can see that tapply() is just the combination of split() and sapply():

```
tapply2 <- function(x, group, f, ..., simplify = TRUE) {
pieces <- split(x, group)
sapply(pieces, f, simplify = simplify)
}
tapply2(pulse, group, length)
#> A B
#> 10 12
tapply2(pulse, group, mean)
#> A B
#> 70.50 74.67
```

The plyr package

- One challenge with using the base functionals is that they have grown organically over time, and have been written by multiple authors. This means that they are not very consistent. For example,
  - The simplify argument is called simplify in tapply() and sapply(), but SIMPLIFY for mapply(), and apply() lacks the argument altogether.
  - vapply() is a variant of sapply() that allows you to describe what the output should be, but there are no corresponding variants of tapply(), apply(), or Map().
  - The first argument to most functionals is the vector, but the first argument to Map() is the function.

- ▷ Additionally, if you think about the combination of input and output types, base R only provides a partial set of functions. This was one of the driving forces behind the creation of the plyr package, which provides consistently named functions with consistently named arguments and implements all combinations of input and output data structures:

	list	**data frame**	**array**
list	llply	ldply	laply
data frame	dlply	ddply	daply
array	alply	adply	aaply

- ▷ Each of these functions splits up the input, applies a function to each piece and then joins the results back together.
- ▷ Overall, this process is called "**split-apply-combine**", and you can read more about it and plyr in The Split-Apply-Combine Strategy for Data Analysis (http://www.jstatsoft.org/v40/i01/), an open-access article published in the Journal of Statistical Software.

Functional programming

- ◇ Another way of thinking about functionals is as a set of general tools for altering, subsetting and collapsing lists.
- ◇ Every functional programming has three tools for this: Map(), Reduce(), and Filter().
- ◇ **Reduce()**
  - ▷ Reduce() recursively reduces a vector, x, to a single value by recursively calling a function f with two arguments at a time.
  - ▷ It combines the first two elements with f, then combines the result of that call with the third element, and so on.
  - ▷ Reduce is also known as **fold**, because it folds together adjacent elements in the list.

    Reduce('+', 1:3) ((1 + 2) + 3)

    Reduce(sum, 1:3) sum(sum(1, 2), 3)
  - ▷ the essence of Reduce() can be described by a simple for loop[47]:

```
Reduce2 <- function(f, x) {
out <- x[[1]]
for(i in seq(2, length(x))) {
out <- f(out, x[[i]])
}
out
}
```

  - ▷ Reduce is an elegant way of turning binary functions into functions that can deal with any number of arguments.
  - ▷ It's useful for implementing many types of recursive operations, like merges and intersections. For example, imagine you had a list of numeric vectors, and you wanted to find the values that occurred in every element:

```
l <- replicate(5, sample(1:10, 15, rep = T), simplify = FALSE)
l
```

[47]The real Reduce() is more complicated because it includes arguments to control whether the values are reduced from the left or from the right (right), an optional initial value (init), and an option to output every intermediate result (accumulate).

```
#> [[1]]
#> [1] 5 8 4 3 5 4 3 6 5 3 7 3 8 10 10
#>
#> [[2]]
#> [1] 2 2 3 4 9 3 5 6 5 2 5 9 7 4 9
#>
#> [[3]]
#> [1] 1 8 6 9 8 7 8 4 10 10 9 4 10 7 1
#>
#> [[4]]
#> [1] 1 1 1 5 2 6 8 10 5 9 4 1 3 1 9
#>
#> [[5]]
#> [1] 2 8 3 4 1 1 8 3 1 2 8 4 4 5 10
```

You could do that by intersecting each element in turn:

```
intersect(intersect(intersect(intersect(l[[1]], l[[2]]), l[[3]]), l[[4]]), l[[5]])
#> [1] 4
```

That's hard to read because of the dagwood sandwich problem, and is equivalent to:

```
Reduce(intersect, l)
#> [1] 4
```

- **Predicate functionals**
  - A **predicate** is a function that returns a single TRUE or FALSE, like is.character, all, or is.NULL. is.na isn't a predicate function because it returns a vector of values.
  - There are three useful predicate functionals in base R: Filter(), Find() and Position().
    - Filter: returns a new vector containing only elements where the predicate is TRUE.
    - Find(): return the first element that matches the predicate (or the last element if right = TRUE).
    - Position(): return the position of the first element that matches the predicate (or the last element if right = TRUE).
  - Another useful functional makes it easy to generate a logical vector from a list (or a data frame) and a predicate:
    ```
    where <- function(f, x) {
    vapply(x, f, logical(1))
    }
    ```
  - compact <- function(x) **Filter**(function(y) !is.null(y), x)
    It removes all null elements from a list

Mathematical functionals

- Functionals are very common in mathematics. The limit, the maximum, the roots (the set of points where f(x) = 0), and the definite integral are all functionals: given a function, they return a single number (or a vector of numbers).
- There are three functions that work with functions that return a single numeric value:

  - ▷ integrate: find the area under the curve given by f
  - ▷ uniroot: find where f hits zero
  - ▷ optimise: find location of lowest (or highest) value of f

- ◇ Maximum likelihood estimation (MLE) is a natural fit for functional programming because we have a well defined problem domain and a general technique to solve it.

- ◇ In MLE, we have two sets of parameters: the data, which is fixed for a given problem, and the parameters, which will vary as we try to find the maximum. That fits naturally with closures because we can have two layers of parameters to a closure.

  Example: Poisson distribution

```
poisson_nll <- function(x) {
n <- length(x)
function(lambda) {
n * lambda - sum(x) * log(lambda) # + terms not involving lambda
}
}
nll1 <- poisson_nll(c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38))
optimise(nll1, c(0, 100))$minimum
#> [1] 32.1
```

- ◇ Another important mathematical functional is optim(). It is a generalisation of optimise() to more than one dimension.

- ◇ If you're interested in how optim() works, you might want to explore the Rvmmin package, which provides a pure-R implementation of optim().

- ◇ Challenge: read about the **fixed point algorithm**. Complete the exercises using R. http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-12.html#%_sec_1.3

Converting loops to functionals, and when it's not possible

- ◇ Sometimes it's possible to torture your code to make it work, but it's usually not a good idea: for loops are verbose and not very expressive, but all R programmers are familiar with them.

- ◇ three types of loop that you shouldn't try and convert into a functional:

  - ▷ modifying in place
  - ▷ recursive functions
  - ▷ while loops

- ◇ Stackoverflow is a good resource for learning more about converting for loops to use functionals. A couple of questions and answers that are particularly helpful are:

  - ▷ "Alternative to loops in R" http://stackoverflow.com/a/14520342/16632
  - ▷ "Speed up the loop operation in R"http://stackoverflow.com/a/2970284/16632

- ◇ **Modifying in place**

- If you need to modify part of an existing data frame, it's often better to use a for loop.

```
trans <- list(
disp = function(x) x * 0.0163871,
am = function(x) factor(x, levels = c("auto", "manual"))
)
for(var in names(trans)) {
mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

- We couldn't normally use lapply() to replace this loop directly, but it is possible to replace the loop with lapply() by using <<-:

```
lapply(names(trans), function(var) {
mtcars[[var]] <<- trans[[var]](mtcars[[var]])
})
```
[48]

◇ Recursive functions

- Another case where it's hard to convert a for loop into a functional is when the relationship is defined recursively.
- Example:

```
exps <- function(x, alpha) {
s <- numeric(length(x) + 1)
for (i in seq_along(s)) {
if (i == 1) { s[i] <- x[i]
}
else {
s[i] <- alpha * x[i - 1] + (1 - alpha) * s[i - 1] }
}
s
}
x <- runif(10)
exps(x, 0.5)
#> [1] 0.8535 0.8535 0.4356 0.6452 0.4794 0.5209 0.5795 0.5397 0.6378 0.7732
#> [11] 0.4325
```

We can't eliminate the for loop because none of the functionals we've seen allow the output at position i to depend on the input and output at position i - 1.

- Another solution for eliminate the for loop in these cases is to solve the **recurrence relation** (http://en.wikipedia.org/wiki/Recurrence_relation#Solving), removing the recursion and replacing it with explicit references. This requires a new set of tools, and is mathematically challenging, but it can pay off by producing a simpler function.

◇ While loops

- the while loop: this keeps running code until a condition is met.

[48] to understand what mtcars[[var]] <<- ... does, you have to understand not only how <<- works, but also what x[[y]] <<- z does behind the scenes.

- ▷ while loops are more general than for loops because you can rewrite every for loop as a while loop, but you can't do the opposite.
- ▷ Not every while loop can be turned into a for loop, because many while loops don't know in advance how many times they will be run:
  ```
  i <- 0
  while(TRUE) {
  if (runif(1) > 0.9) break
  i <- i + 1
  }
  ```
  [49]
- ▷ This is a common situation when you're writing simulations: one of the random parameters in your simulation may be how many times a process occurs.

A family of functions (case study)

1. start by defining a very simple plus function, that takes two scalar arguments:
   ```
   add <- function(x, y) {
   stopifnot(length(x) == 1, length(y) == 1,
   is.numeric(x), is.numeric(y))
   x + y
   }
   ```
   [50]
2. We really should also have some way to deal with missing values. A helper function will make this a bit easier: if x is missing it should return y, if y is missing it should returns x, and if both x and y are missing then it should returns another argument to the function: identity.
   ```
   rm_na <- function(x, y, identity) {
   if (is.na(x) && is.na(y)) {
   identity
   }
   else if (is.na(x)) {
   y
   }
   else {
   x
   }
   }
   rm_na(NA, 10, 0)
   #> [1] 10
   rm_na(10, NA, 0)
   #> [1] 10
   rm_na(NA, NA, 0)
   #> [1] 0
   ```

[49] This is equivalent to counting how many times a Bernoulli trial with p = 0.1 is run before it is successful: this is a geometric random variable so you could replace the above code with i <- rgeom(1, 0.1).

[50] We're using R's existing addition operator here, which does much more, but the focus in this section is on how we can take very very simple functions and extend them to do more

3. That allows us to write a version of add that can deal with missing values if needed: (and it often is!)

   ```
   add <- function(x, y, na.rm = FALSE) {
   if (na.rm && (is.na(x) || is.na(y))) rm_na(x, y, 0) else x + y }
   add(10, NA)
   #> [1] NA
   add(10, NA, na.rm = TRUE)
   #> [1] 10
   add(NA, NA)
   #> [1] NA
   add(NA, NA, na.rm = TRUE)
   #> [1] 0
   ```

4. Now we have the basics working, we can extend this function to deal with more complicated inputs. The first way we might want to extend it is add more than two numbers together. This is a simple application of Reduce: if the input is c(1, 2, 3), then we want to compute add(1, add(2, 3)):

   ```
   r_add <- function(xs, na.rm = TRUE) {
   Reduce(function(x, y) add(x, y, na.rm = na.rm), xs) }
   r_add(c(1, 4, 10))
   #> [1] 15
   ```

   However:

   ```
   r_add(NA, na.rm = TRUE)
   #> [1] NA
   r_add(numeric())
   #> NULL
   ```

   - ◊ These are incorrect: in the first case we get a missing value even thought we've explicitly asked for them to be ignored, and in the second case we get NULL, instead of a length 1 numeric vector (as for every other set of inputs).
   - ◊ The two problems are related: if we give Reduce() a length one vector it doesn't have anything to reduce, so it just returns the input; if we give it a length 0 input it always returns NULL.
   - ◊ There are two ways to fix this: we can concatenate 0 to every input vector, or we can use the init argument to Reduce() (which effectively does the same thing):

     ```
     r_add <- function(xs, na.rm = TRUE) {
     Reduce(function(x, y) add(x, y, na.rm = na.rm), c(0, xs))
     }
     r_add(c(1, 4, 10))
     #> [1] 15
     r_add(NA, na.rm = TRUE)
     #> [1] 0
     r_add(numeric())
     #> [1] 0
     ```
     [51]

[51](This is equivalent to sum())

5. It would also be nice to have a vectorised version of add so that we can give it two vectors of numbers to add in parallel. We have two ways, using Map() or vapply(), to implement this:

```
v_add <- function(x, y, na.rm = TRUE) {
stopifnot(length(x) == length(y),
is.numeric(x), is.numeric(y)
)
Map(function(x, y) add(x, y, na.rm = na.rm), x, y)
}
v_add <- function(x, y, na.rm = TRUE) {
stopifnot(length(x) == length(y),
is.numeric(x), is.numeric(y)
)
vapply(seq_along(x), function(i) add(x[i], y[i], na.rm = na.rm),
numeric(1))
}
v_add(1:10, 1:10)
#> [1] 2 4 6 8 10 12 14 16 18 20
v_add(numeric(), numeric())
#> numeric(0)
v_add(c(1, NA), c(1, NA))
#> [1] 2 0
v_add(c(1, NA), c(1, NA), na.rm = TRUE)
#> [1] 2 0
```

6. Another variant of adding is the cumulative sum: it's like the reductive version, but we see every step along the way to the final result. This is easy to implement with Reduce()'s accumuate argument:

```
c_add <- function(xs, na.rm = FALSE) {
Reduce(function(x, y) add(x, y, na.rm = na.rm), xs, accumulate = TRUE) }
c_add(1:10)
#> [1] 1 3 6 10 15 21 28 36 45 55
c_add(10:1)
#> [1] 10 19 27 34 40 45 49 52 54 55
```

7. Finally, we might want to define versions for more complicated data structures like matrices. We could create row and col variants that sum across rows and columns respectively, or we could go the whole hog and define an array version that would sum across any arbitrary dimensions of an array. These are easy to implement: they're a combination of add() and apply()

```
row_sum <- function(x, na.rm = TRUE) apply(x, 1, add, na.rm = na.rm)
col_sum <- function(x, na.rm = TRUE) apply(x, 2, add, na.rm = na.rm)
arr_sum <- function(x, dim, na.rm = TRUE) apply(x, dim, add, na.rm = na.rm)
```
[52]

[52](These are equivalent to rowSums() and colSums())

- ◇ There are two main reasons to create our own functions although they're existing in base R:
  1. we've created all our variants from a very simple binary operator (add) and a well-tested functional (Reduce, Map and apply), so we know all the variants will behave consistently.
  2. we've seen the infrastructure for addition, so we can now adapt it to other operators that might not have the full suite variants in base R.
- ◇ The downside of this approach is that these implementations are unlikely to be efficient (For example, colSums(x) is much faster than apply(x, 2, sum)).
- ◇ External resource: List out of lambda (`http://stevelosh.com/blog/2013/03/list-out-of-lambda/`)[53]

3.3 Function operators

- ◇ **Function operators**: functions that take one (or more) functions as input and return a function as output.
- ◇ Function operators are a FP technique related to functionals, but where functionals abstract away common uses of loops, function operators abstract over common uses of anonymous functions.
- ◇ Like functionals, there's nothing you can't do without them; but they can make your code more readable, more expressive and faster to write.
- ◇ we'll start to build up tools that replace standard anonymous functions with specialised equivalents that allow us to communicate our intent more clearly.
- ◇ learn about partial application and the partial() function. Partial application encapsulates the use of an anonymous function to supply default arguments, and leads to the succinct code.
- ◇ an important use of FOs: you can eliminate parameters to a functional by instead transforming the input function. This approach allows your functionals to be more extensible: as long as the inputs and outputs of the function remain the same, your functional can be extended in ways you haven't thought of.
- ◇ To explore four types of function operators (FOs). Function operators can:
  1. **add behaviour** while leaving the function otherwise unchanged, like automatically logging when the function is run, ensuring a function is run only once, or delaying the operation of a function.
  2. **change output**, for example by returning a value if the function throws an error, or negating the result of a logical predicate.
  3. **change input**, like partially evaluating the function, converting a function that takes multiple arguments to a function that takes a list, or automatically vectorising a function.
  4. **combine functions**, for example, combining the results of predicate functions with boolean operators, or composing multiple function calls.
- ◇ **In other languages**

  Function operators are used extensively in FP languages like Haskell, and are common in Lisp, Scheme and Clojure. They are an important part of modern JavaScript programming, like in the underscore.js library, and are particularly common in CoffeeScript, since the syntax for anonymous functions is so concise. Stack based languages like Forth and Factor use function operators almost exclusively, since

[53] a blog article by Steve Losh that explores how you can produces higher level language structures (like lists) out of more primitive language features (like closures, aka lambdas)

it is rare to refer to variables by name. Python's decorators are just function operators by a different name. They are very rare in Java, because it's difficult to manipulate functions (although possible if you wrap them up in strategy-type objects), and also rare in C++; while it's possible to create objects that work like functions ("functors") by overloading the () operator, modifying these objects with other functions is not a common programming technique. That said, C++ 11 adds partial application (std::bind) to the standard library.

Behavioural FOs

- ♢ The first class of FOs are those that leave the inputs and outputs of a function unchanged, but add some extra behaviour.
- ♢ Example:
  - ▷ imagine we want to download a long vector of urls with download.file(). That's pretty simple with lapply():
    ```
    lapply(urls, download.file, quiet = TRUE)
    ```
    [54]
  - ▷ Because we have a long list we want to print some output so that we know it's working (we'll print a . every ten urls), and we also want to avoid hammering the server, so we add a small delay to the function between each call. That leads to a rather more complicated for loop (we can no longer use lapply() because we need an external counter):
    ```
    i <- 1 for(url in urls) {
    i <- i + 1
    if (i %% 10 == 0) cat(".")
    Sys.delay(1)
    download.file(url, quiet = TRUE)
    }
    ```
  - ▷ Reading this code is quite hard because we are using low-level functions, and it's not obvious (without some thought), what the overall objective is.
- ♢ **Useful behavioural FOs**
  - ▷ Function delay_by():
    ```
    delay_by <- function(delay, f) {
    function(...) {
    Sys.sleep(delay)
    f(...)
    }
    }
    system.time(runif(100))
    #> user system elapsed
    #> 0.000 0.000 0.001
    system.time(delay_by(1, runif)(100))
    #> user system elapsed
    #> 0.000 0.001 1.001
    ```

[54]This example ignores the fact that download.file also needs a file name, so pretend it has a useful default for the purposes of this exposition.

- ▷ Function dot_every()[55]: more complicated because it needs to modify state in the parent environment using <<-.

```
dot_every <- function(n, f) {
i <- 1
function(...) {
if (i %% n == 0) cat(".")
i <<- i + 1 f(...)
}
}
x <- lapply(1:100, runif)
x <- lapply(1:100, dot_every(10, runif))
#> ..........
```

- ▷ Two other tasks that we can solve with a behaviour FO are:

  1. Logging a time stamp and message to a file everytime a fnction is run

```
log_to <- function(path, message, f) {
stopifnot(file.exists(path))
function(...) {
cat(Sys.time(), ": ", message, sep = "", file = path, append = TRUE)
f(...)
}
}
```

  2. Ensuring that if the first input is NULL then the output is NULL

```
maybe <- function(f) {
function(x, ...) {
if (is.null(x)) return(NULL)
f(x, ...)
}
}
```

◇ **Memoisation**

- ▷ Another thing you might worry about when downloading multiple files is accidentally downloading the same file multiple times.
- ▷ You could avoid it by calling unique on the list of input urls, or manually managing a data structure that mapped the url to the result.
- ▷ An alternative approach is to use memoisation: a way of modifying a function to automatically cache its results.

```
library(memoise)
slow_function <- function(x) {
Sys.sleep(1)
10
}
system.time(slow_function())
#> user system elapsed
#> 0.000 0.001 1.000
```

[55]Notice that I've made the function the last argument to each FO. This makes it read a little better when we compose multiple function operators.

```
system.time(slow_function())
#> user system elapsed
#> 0 0 1
fast_function <- memoise(slow_function)
system.time(fast_function())
#> user system elapsed
#> 0.000 0.001 1.001
system.time(fast_function())
#> user system elapsed
#> 0 0 0
```

- ▷ Memoisation is an example of a classic tradeoff in computer science: trading space for speed.[56]
- ▷ It is useful when the computation we do is recursively and depends on the previous results the function's calculated.
- ▷ A somewhat more realistic use case is implementing the Fibonacci series. The Fibonacci series is defined recursively: the first two values are 1 and 1, then f(n) = f(n - 1) + f(n - 2). Memoising fib() makes the implementation much faster because each value is only computed once, and then remembered.

```
fib <- function(n) {
if (n < 2) return(1)
fib(n - 2) + fib(n - 1)
}
system.time(fib(23))
#> user system elapsed
#> 0.220 0.007 0.227
system.time(fib(24))
#> user system elapsed
#> 0.344 0.000 0.344
fib2 <- memoise(function(n) {
if (n < 2) return(1)
fib2(n - 2) + fib2(n - 1)
})
system.time(fib2(23))
#> user system elapsed
#> 0.004 0.000 0.004
system.time(fib2(24))
#> user system elapsed
#> 0.001 0.000 0.001
```

- ▷ It doesn't make sense to memoise all functions. The example below shows that a memoised random number generator is no longer random:

```
runifm <- memoise(runif)
runifm(5)
#> [1] 0.9671 0.7691 0.8748 0.6358 0.5873
runifm(5)
#> [1] 0.9671 0.7691 0.8748 0.6358 0.5873
```

[56] A memoised function uses more memory (because it stores all of the previous inputs and outputs), but is much faster.

- ▷ download <- dot_every(10, memoise(delay_by(1, download.file)))

◇ **Capturing function invocations**

- ▷ One challenge with functionals is that it can be hard to see what's going on - it's not easy to pry open the internals like it is with a for loop.
- ▷ The tee function, defined below, has three arguments, all functions: f, the original function; on_input, a function that's called with the inputs to f, and on_output a function that's called with the output from f.

```
ignore <- function(...) NULL
tee <- function(f, on_input = ignore, on_output = ignore) {
function(...) {
input <- if (nargs() == 1) c(...) else list(...)
on_input(input)
output <- f(...)
on_output(output)
output
}
}
```

- ▷ We can use tee to look into how uniroot finds where x and cos(x) intersect:

```
g <- function(x) cos(x) - x
zero <- uniroot(g, c(-5, 5))
# The location where the function is evaluated, i.e. root
zero <- uniroot(tee(g, on_input = print), c(-5, 5))
#> [1] -5
#> [1] 5
#> [1] 0.283662
#> [1] 0.875203
#> [1] 0.72298
#> [1] 0.738631
#> [1] 0.739085
#> [1] 0.739024
#> [1] 0.739085
# The value of the function
zero <- uniroot(tee(g, on_output = print), c(-5, 5))
#> [1] 5.28366
#> [1] -4.71634
#> [1] 0.676375
#> [1] -0.234363
#> [1] 0.0268568
#> [1] 0.00076012
#> [1] -0.000000260399
#> [1] 0.000101887
#> [1] -0.000000260399
```

- ▷ We might want to capture the sequence of the calls. To do that we create a function called remember() that remembers every argument it was called with, and retrieves them when coerced into a list.

```
remember <- function() {
memory <- list()
f <- function(...) {
# This is inefficient!
memory <<- append(memory, list(...))
invisible()
}
structure(f, class = "remember")
}
as.list.remember <- function(x, ...) {
environment(x)$memory
}
print.remember <- function(x, ...) {
cat("Remembering...\n")
str(as.list(x))
}
```

- ▷ Now we can see exactly how uniroot zeros in on the final answer:

```
locs <- remember()
vals <- remember()
zero <- uniroot(tee(g, locs, vals), c(-5, 5))
x <- sapply(as.list.remember(locs), "[[", 1)
error <- sapply(as.list.remember(vals), "[[", 1)
plot(x, type = "b"); .abline(h = 0.739, col = "grey50")
```

Output FOs

- ◇ This could be quite simple, or it could fundamentally change the operation of the function, returning something completely different to its usual output.
- ◇ **Minor modifications**
  - ▷ base::Negate and plyr::failwith offer two minor, but useful, modifications of a function that are particularly handy in conjunction with functionals.
  - ▷ Negate takes a function that returns a logical vector (a predicate function), and returns the negation of that function.

```
Negate <- function(f) {
function(...) !f(...) }
(Negate(is.null))(NULL)
#> [1] FALSE
```

  - ▷ Application: function to remove all null elements from a list

```
compact <- function(x) Filter(Negate(is.null), x)
```

- plyr::failwith() turns a function that throws an error into a function that returns a default value when there's an error.

```
failwith <- function(default = NULL, f, quiet = FALSE) {
function(...) {
out <- default
try(out <- f(...), silent = quiet)
out
}
} l
og("a")
#> Error: non-numeric argument to mathematical function
failwith(NA, log)("a")
#> [1] NA
failwith(NA, log, quiet = TRUE)("a")
#> [1] NA
```

- failwith() is very useful in conjunction with functionals: instead of the failure propagating and terminating the higher-level loop, you can complete the iteration and then find out what went wrong.
- imagine you're fitting a set of generalised linear models (glms) to a list of data frames. Sometimes glms fail because of optimisation problems. You still want to try to fit all the models, then once that's complete, look at the data sets that failed to fit:

```
# If any model fails, all models fail to fit:
models <- lapply(datasets, glm, formula = y ~ x1 + x2 * x3)
# If a model fails, it will get a NULL value
models <- lapply(datasets, failwith(NULL, glm), formula = y ~ x1 + x2 * x3)
# remove failed models (NULLs) with compact
ok_models <- compact(models)
# use where to extract the datasets corresponding to failed models
failed_data <- datasets[vapply(models, is.null, logical(1))]
```

◇ **Changing what a function does**

- Instead of returning the original return value, we can return some other effect of the function evaluation.

1. Return text that the function print()ed:

```
capture_it <- function(f) {
function(...) {
capture.output(f(...))
}
}
str_out <- capture_it(str)
str(1:10)
#> int [1:10] 1 2 3 4 5 6 7 8 9 10
str_out(1:10)
#> [1] " int [1:10] 1 2 3 4 5 6 7 8 9 10"
```

2. Return how long a function took to run:
```
time_it <- function(f) {
function(...) {
system.time(f(...))
}
}
```
Then we can use this function to revise the functional in previous chapter:
```
compute_mean <- list(
base = function(x) mean(x),
sum = function(x) sum(x) / length(x) ) x <- runif(1e6)
# Instead of using an anonymous function to time
lapply(compute_mean, function(f) system.time(f(x)))
#> $base
#> user system elapsed
#> 0.007 0.000 0.007
#>
#> $sum
#> user system elapsed
#> 0.004 0.000 0.005
# We can compose function operators
call_fun <- function(f, ...) f(...)
lapply(compute_mean, time_it(call_fun), x) [57]
#> $base
#> user system elapsed
#> 0.007 0.000 0.008
#>
#> $sum
#> user system elapsed
#> 0.004 0.000 0.004
```

▷ In this example, there's not a huge benefit to using function operators, because the composition is simple and we're applying the same operator to each function. Generally, using function operators is more effective when you are using multiple operators or if the gap between creating them and using them is large.

Input FOs

◇ Prefilling function arguments: partial function application

▷ A common use of anonymous functions is to make a variant of a function that has certain arguments "filled in" already. This is called "**partial function application**", and is implemented by pryr::partial.

▷ **partial()** allows us to replace code like
```
f <- function(a) g(a, b = 1)
compact <- function(x) Filter(Negate(is.null), x)
Map(function(x, y) f(x, y, zs), xs, ys)
```
with
```
f <- partial(g, b = 1)
compact <- partial(Filter, Negate(is.null))
Map(partial(f, zs = zs), xs, ys)
```

[57] Can not directly use time_it in lapply, which will cause error, instead we need to use call_fun to help.

- We can use this idea to simplify some of the code we used when working with lists of functions. Instead of:
  ```
  funs2 <- list(
  sum = function(x, ...) sum(x, ..., na.rm = TRUE),
  mean = function(x, ...) mean(x, ..., na.rm = TRUE),
  median = function(x, ...) median(x, ..., na.rm = TRUE) )
  ```
  We can write:
  ```
  library(pryr)
  funs2 <- list(
  sum = partial(sum, na.rm = TRUE),
  mean = partial(mean, na.rm = TRUE),
  median = partial(median, na.rm = TRUE) )
  ```
- The repeating pattern allows us to reduce the code still further:
  ```
  funs <- c(sum = sum, mean = mean, median = median)
  funs2 <- lapply(funs, partial, na.rm = TRUE)
  ```
- Let's think about a similar, but subtly different case. Say we have a numeric vector and we want to generate a list of trimmed means with that amount of trimming.
  - Doesn't work: the same trims vector is assigned to every mean function
    ```
    (trims <- seq(0, 0.9, length = 5))
    #> [1] 0.000 0.225 0.450 0.675 0.900
    funs3 <- lapply(trims, partial, '_f' = mean)
    sapply(funs3, call_fun, c(1:100, (1:50) * 100))
    #> Error: 'trim' must be numeric of length one
    ```
  - Instead we could use an anonymous function, but still doesn't work because each function gets a promis to evaluate t, and that promise isn't evaluated until all of the functions are run, by which time t=0.9.
    ```
    funs4 <- lapply(trims, function(t) partial(mean, trim = t))
    funs4[[1]]
    #> function (...)
    #> mean(trim = t, ...)
    #> <environment: 0x48aebc0>
    sapply(funs4, call_fun, c(1:100, (1:50) * 100))
    #> [1] 75.5 75.5 75.5 75.5 75.5
    ```
  - To make it work you need to manually force the evaluation of t:
    ```
    funs5 <- lapply(trims, function(t) {
    force(t)
    partial(mean, trim = t)
    })
    funs5[[1]]
    #> function (...)
    #> mean(trim = t, ...)
    #> <environment: 0x5238158>
    sapply(funs5, call_fun, c(1:100, (1:50) * 100))
    #> [1] 883.7 235.6 75.5 75.5 75.5
    ```
- When writing functionals, you can expect your users to know of **partial()** and not use **...** For example, instead of implementing **lapply()** like:
  ```
  lapply2 <- function(x, f, ...) {
  ```

```
out <- vector("list", length(x))
for (i in seq_along(x)) {
out[[i]] <- f(x[[i]], ...)
}
out
}
unlist(lapply2(1:5, log, base = 10))
#> [1] 0.0000 0.3010 0.4771 0.6021 0.6990
```

we could implement it as:

```
lapply3 <- function(x, f) {
out <- vector("list", length(x))
for (i in seq_along(x)) {
out[[i]] <- f(x[[i]]) }
out
}
unlist(lapply3(1:5, partial(log, base = 10)))
#> [1] 0.0000 0.3010 0.4771 0.6021 0.6990
```

▷ Partial function application is straightforward in many functional programming languages, but it's not entirely clear how it should interact with R's lazy evaluation rules. The approach plyr::partial takes is to create a function as similar as possible to the anonymous function you'd create by hand. Peter Meilstrup takes a different approach in his ptools package (https://github.com/crowding/ptools/); you might want to read about %()%, %>>% and %<<% if you're interested in the topic.

◇ Changing input types

▷ Instead of a minor change to the function's inputs, it's also possible to make a function work with a fundamentally different type of data. There are a few existing functions along these lines:

1. base::Vectorize converts a scalar function to a vector function. Vectorize takes a non-vectorised function and vectorises with respect to the arguments given in the vectorizge.args parameter.
   - ◇ A mildly useful extension of sample would be to vectorize it with respect to size: this would allow you to generate multiple samples in one call.
     sample2 <- Vectorize(sample, "size", SIMPLIFY = FALSE) [58]
     sample2(1:5, c(1, 1, 3))[59]
2. splat converts a function that takes multiple arguments to a function that takes a single list of arguments.
   ```
   splat <- function (f) {
   function(args) {
   do.call(f, args)
   }
   }
   ```
   - ◇ This is useful if you want to invoke a function with varying arguments:
     ```
     x <- c(NA, runif(100), 1000)
     args <- list(
     list(x), list(x, na.rm = TRUE), list(x, na.rm = TRUE, trim = 0.1) ) lapply(args, splat(mean))
     #> [[1]] #> [1] NA #> #> [[2]] #> [1] 10.38 #> #> [[3]] #> [1] 0.4897
     ```

[58] In this example we have used SIMPLIFY = FALSE to ensure that our newly vectorised function always returns a list. This is usually what you want.

[59] This will generate 3 samples with length 1, 1 and 3 respectively.

3. **plyr::colwise()** converts a vector function to one that works with data frames:
```
summaries <- plyr::each(mean, sd, median)
summaries(1:10)
#> mean sd median
#> 5.500 3.028 5.500
```

Combining FOs

- ♢ Instead of operating on single functions, function operators can take multiple functions as input.
- ♢ One simple example of this is plyr::each() which takes a list of vectorised functions and returns a single function that applies each in turn to the input:
- ♢ These are glue that join multiple functions together.
- ♢ **Function composition**
  - ▷ An important way of combining functions is through composition: f(g(x)). Composition takes a list of functions and applies them sequentially to the input.
  - ▷ It's a replacement for the common anonymous function pattern where you chain together multiple functions to get the result you want:
    ```
    sapply(mtcars, function(x) length(unique(x)))
    #> mpg cyl disp hp drat wt qsec vs am gear carb
    #> 25 3 27 22 22 29 30 2 2 3 6
    ```
  - ▷ pryr::compose() provides a fuller-featured alternative that can accept multiple functions:
    ```
    function (...) {
    fs <- lapply(list(...), match.fun)
    n <- length(fs)
    last <- fs[[n]]
    rest <- fs[-n]
    function(...) {
    out <- last(...)
    for (f in rev(rest)) {
    out <- f(out)
    }
    out
    }
    }
    ```
  - ▷ This allows us to write:
    ```
    sapply(mtcars, compose(length, unique))
    #> mpg cyl disp hp drat wt qsec vs am gear carb
    #> 25 3 27 22 22 29 30 2 2 3 6
    ```
  - ▷ Mathematically, function composition is often denoted with an infix operator, o, (f o g)(x).
  - ▷ In R, we can create our own infix function that works similarly:
    ```
    "%.%" <- compose
    sapply(mtcars, length %.% unique)
    #> mpg cyl disp hp drat wt qsec vs am gear carb
    ```

```
#> 25 3 27 22 22 29 30 2 2 3 6
sqrt(1 + 8)
#> [1] 3
compose(sqrt, '+')(1, 8)
#> [1] 3
(sqrt %.% '+')(1, 8)
#> [1] 3
```

- ▷ compose() is particularly useful in conjunction with partial(), because partial() allows you to supply additional arguments to the functions being composed.
- ▷ Compose also allows for a very succinct implement of Negate: it's just a partially evaluated version of compose().
  `Negate <- partial(compose, '¡)`
- ▷ We could also implement the standard deviation by breaking it down into a separate set of function compositions:

```
square <- function(x) x ^ 2
deviation <- function(x) x - mean(x)
sd <- sqrt %.% mean %.% square %.% deviation sd(1:10)
#> [1] 2.872
```

- ▷ This type of programming is called **tacit** or **point-free programming**. (The term point free comes from the use of the word point to refer values in topology; this style is also derogatorily known as **pointless**).[60]
- ▷ One nice side effect of this style of programming is that it keeps the arguments to each function near the function name. This is important because code gets harder to understand as the size of the chunk of code you have to hold in your head grows.
- ▷ Two new versions based on compose and partial:

```
download <- dot_every(10, memoise(delay_by(1, download.file)))
download <- pryr::compose(
partial(dot_every, 10),
memoise,
partial(delay_by, 1),
download.file
)
download <- partial(dot_every, 10) %.%
memoise %.%
partial(delay_by, 1) %.%
download.file
```

◊ Logical predicates and boolean algebra

- ▷ When I use Filter() and other functionals that work with logical predicates, we often anonymous functions to combine multiple conditions:
  `Filter(function(x) is.character(x) || is.factor(x), iris)`

[60] In this style of programming you don't explicitly refer to variables, focussing on the high-level composition of functions, rather than the low-level flow of data. Since we're using only functions and not parameters, we use verbs and not nouns, and this style leads to code that focusses on what's being done, not what it's being done to. This style is common in Haskell, and is the typical style in stack based programming languages like Forth and Factor. It's not a terribly natural or elegant style in R, but it is a useful tool to have in your toolbox.

▷ As an alternative, we could define some function operators that combine logical predicates:

```
and <- function(f1, f2) {
function(...) { f1(...) && f2(...)
}
}
or <- function(f1, f2) {
function(...) {
f1(...) || f2(...)
}
}
not <- function(f1) {
function(...) {
!f1(...)
}
}
```

which would allow us to write:

```
Filter(or(is.character, is.factor), iris)
```

The common pattern and a subtle bug

◇ Most function operators we've seen follow a similar pattern:

```
funop <- function(f, otherargs) {
function(...) {
# maybe do something
res <- f(...)
# maybe do something else
res
}
}
```

◇ There's a subtle problem with this implementation. It does not work well with lapply() because f is lazily evaluated.

◇ This means that if you give lapply() a list of functions and a FO to apply it to each of them, it will look like it repeatedly applied the FO to the last function:

```
wrap <- function(f) {
function(...) f(...)
}
fs <- list(sum = sum, mean = mean, min = min)
gs <- lapply(fs, wrap)
gs$sum(1:10)
#> [1] 1
environment(gs$sum)$f
#> function (..., na.rm = FALSE) .Primitive("min")
```

- ◊ Another problem is that as designed, we have to pass in a funtion object, not the name of a function, which is often convenient.
- ◊ We can solve both problems by using match.fun(): it forces evaluation of f, and will find the function object if given its name:

```
wrap2 <- function(f) {
f <- match.fun(f)
function(...) f(...)
}
fs <- c(sum = "sum", mean = "mean", min = "min")
hs <- lapply(fs, wrap2)
hs$sum(1:10)
#> [1] 55
environment(hs$sum)$f
#> function (..., na.rm = FALSE) .Primitive("sum")
```

4 Metaprogramming[61]

4.1 Introduction

◇ In R you can also access the expression used to compute them. Combined with R's lazy evaluation mechanism this gives function authors considerable power to both access the underlying expression and do special things with it.

◇ Techniques based on these tools are generally called "**computing on the language**", and in R provide a set of tools with power equivalent to functional and object oriented programming.

Capturing expressions

◇ The tool that makes non-standard evaluation possible in R is substitute(). It looks at a function argument, and instead of seeing the value, it looks to see how the value was computed:

```
f <- function(x) {
substitute(x)
}
f(1:10)
#> 1:10
f(x)
#> x
f(x + y ^ 2 / z + exp(a * sin(b)))
#> x + y^2/z + exp(a * sin(b))
```

◇ substitute() works because function arguments in R are only evaluated when they are needed, not automatically when the function is called.

◇ This means that function arguments are not just a simple value, but instead store both the expression to compute the value and the environment in which to compute it.[62]

◇ deparse() function takes an expression and converts it to a character vector.

```
g <- function(x)
deparse(substitute(x))
g(1:10)
#> [1] "1:10"
g(x)
#> [1] "x"
g(x + y ^ 2 / z + exp(a * sin(b)))
#> [1] "x + y^2/z + exp(a * sin(b))"
```

▷ There's one important caveat with deparse(): it can return a multiple strings if the input is long: g(a + b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q + r + s + t + u + v + w + x + y + z)[63]

[61] "Flexibility in syntax, if it does not lead to ambiguity, would seem a reasonable thing to ask of an interactive programming language." — Kent Pitman

[62] Together these two things are called a **promise**.

[63] If you need a single string, you can work around this by using the width.cutoff argument (which has a maximum value of 500), or by joining the lines back together again with paste().

- There are a lot of functions in base R that use these ideas. Some use them to avoid quotes:

  ```
  library(ggplot2)
  library("ggplot2")
  ```

- Others use them to provide default labels.

  ```
  plot.default <- function(x, y = NULL, xlabel = NULL, ylabel = NULL, ...) {
  ...
  xlab <- if (is.null(xlabel) && !missing(x)) deparse(substitute(x))
  ylab <- if (is.null(ylabel) && !missing(y)) deparse(substitute(y))
  ...
  }
  ```
  [64]

- data.frame() does a similar thing. It automatically labels variables with the expression used to compute them:

  ```
  x <- 1:4
  y <- letters[1:4]
  names(data.frame(x, y))
  #> [1] "x" "y"
  ```

Non-standard evaluation in subset

- Subset is special because vs == am or cyl == 4 aren't evaluated in the global environment: instead they're evaluated in the data frame.

  ```
  subset(mtcars, vs == am)
  ```

- In other words, subset() implements different scoping rules so instead of looking for those variables in the current environment, subset() looks in the specified data frame.

- This is called **non-standard evaluation**: you are deliberately breaking R's usual rules in order to do something special.

- To do this we need eval(), which takes an expression and evaluates it in the specified environment.

- quote() is similar to substitute() but it always gives you back exactly the expression you entered. This makes it useful for interactive experimentation.

  ```
  quote(1:10)
  #> 1:10
  quote(x)
  #> x
  quote(x + y ^ 2 / z + exp(a * sin(b)))
  #> x + y^2/z + exp(a * sin(b))
  ```

[64] the real code is more complicated because of the way base plotting methods work, but it's effectively the same

◇ If you only provide one argument, it evaluates the expression in the current environment. This makes eval(quote(x)) exactly equivalent to typing x, regardless of what x is:

```
eval(quote(x <- 1))
eval(quote(x))
#> [1] 1
eval(quote(cyl))
#> Error: object 'cyl' not found
```

◇ Note that quote() and eval() are basically opposites. In the example below, each eval() peels off **(exactly) one** layer of quoting.

```
quote(quote(2 + 2))
#> quote(2 + 2)
eval(quote(quote(2 + 2)))
#> 2 + 2
eval(eval(quote(quote(2 + 2))))
#> [1] 4
```

◇ The second argument to eval() controls which environment the code is evaluated in:

```
x <- 10
eval(quote(x))
#> [1] 10
e <- new.env()
e$x <- 20
eval(quote(x), e)
#> [1] 20
```

◇ Instead of an environment, the second argument can also be a list or a data frame. This works because an environment is basically a set of mappings between names and values, in the same way as a list or data frame.

```
eval(quote(x), list(x = 30))
#> [1] 30
eval(quote(x), data.frame(x = 40))
#> [1] 40
```

◇ This is basically what we want for subset():

```
eval(quote(cyl == 4), mtcars)
#> [1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
#> [12] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE FALSE
#> [23] FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE FALSE TRUE
```

◇ We can combine eval() and substitute() together to write subset(): we can capture the call representing the condition, evaluate it in the context of the data frame, and then use the result for subsetting:

```
subset2 <- function(x, condition) {
```

```
condition_call <- substitute(condition)
r <- eval(condition_call, x)
x[r, ]
}
```

◇ When you first start using eval() it's easy to make mistakes. Here's a common one: forgetting to quote the input:

```
eval(cyl, mtcars)
#> Error: object 'cyl' not found
# Carefully look at the difference to this error
eval(quote(cyl), mtcars)
#> [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

◇ What does evalq() do? Use it to reduce the amount of typing for the examples above that use both eval() and quote()

Scoping issues

◇ If eval() can't find the variable inside the data frame (it's second argument), it's looking in the function environment. That's obviously not what we want, so we need some way to tell eval() to look somewhere else if it can't find the variables in the data frame.

◇ The key is the third argument: enclos. This allows us to specify the parent (or enclosing) environment for objects that don't have one like lists and data frames (enclos is ignored if we pass in a real environment). The enclosing environment is where any objects that aren't found in the data frame will be looked for. By default it uses the environment of the current function, which is not what we want.

◇ We want to look for x in the environment in which subset was called. In R terminology this is called the **parent frame** and is accessed with parent.frame().

```
subset2 <- function(x, condition) {
condition_call <- substitute(condition)
r <- eval(condition_call, x, parent.frame())
x[r, ] }
x <- 4
subset2(mtcars, cyl == x)
```

◇ Using enclos is just a short cut for converting a list or data frame to an environment with the desired parent yourself. We can use the list2env() to turn a list into an environment and explicitly set the parent ourselves:

```
subset2 <- function(x, condition) {
condition_call <- substitute(condition)
env <- list2env(x, parent = parent.frame())
r <- eval(condition_call, env)
x[r, ]
}
x <- 4
subset2(mtcars, cyl == x)
```

- When evaluating code in a non-standard way, it's also a good idea to test your code works when run outside of the global environment:

  ```
  f <- function() {
  x <- 6
  subset(mtcars, cyl == x)
  }
  f()
  ```

- plyr::arrange() works similarly to subset(), but instead of selecting rows, it reorders them.
- What does transform() do?
- plyr::mutate() is similar to transform() but it applies the transformations sequentially so that transformation can refer to columns that were just created:

  ```
  df <- data.frame(x = 1:5)
  transform(df, x2 = x * x, x3 = x2 * x)
  plyr::mutate(df, x2 = x * x, x3 = x2 * x)
  ```

- What does with() do?
- What does within() do?

Calling from another function

- Typically, computing on the language is most useful for functions called directly by the user, not by other functions. While subset saves typing, it has one big disadvantage: it's now difficult to use non-interactively, e.g. from another function.

  ```
  colname <- "cyl"
  val <- 6
  subset2(mtcars, colname == val)
  #> [1] mpg cyl disp hp drat wt qsec vs am gear carb
  #> <0 rows> (or 0-length row.names)
  # Zero rows because "cyl" != 6
  ```

- Or imagine we want to create a function that randomly reorders a subset of the data:

  ```
  subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x, parent.frame())
  x[r, ]
  }
  scramble <- function(x) x[sample(nrow(x)), ]
  subscramble <- function(x, condition) {
  scramble(subset2(x, condition))
  }
  ```

  - ▷ But this doesn't work: condition_call contains the expression condition so when we try to evaluate that it evaluates condition which has the value cyl == 4. This can't be computed in the parent environment because it doesn't contain an object called cyl.

- ♢ This is an example of the general tension between functions that are designed for interactive use, and functions that are safe to program with. A function that uses substitute() might save typing, but it's difficult to call from another function. As a developer you should also provide an alternative version that works when passed a quoted expression. For example, we could rewrite:

```
subset2_q <- function(x, condition) {
r <- eval(condition, x, parent.frame())
x[r, ]
}
subset2 <- function(x, condition) {
subset2_q(x, substitute(condition))
}
subscramble <- function(x, condition) {
condition <- substitute(condition)
scramble(subset2_q(x, condition))
}
```

Substitute

- ♢ Following the examples above, whenever you write your own functions that use non-standard evaluation, you should always provide alternatives that others can use.
- ♢ what happens if you want to call a function that uses non-standard evaluation and doesn't have a form that takes expressions?

```
x <- quote(mpg)
y <- quote(disp)
xyplot(x ~ y, data = mtcars)
```

- ♢ Again, we can turn to substitute and use it for another purpose: modifying expressions.
- ♢ Unfortunately substitute() has a "feature" that makes experimenting with it interactively a bit of a pain: it never does substitutions when run from the global environment, and just behaves like quote():
- ♢ But if we run it inside a function, substitute() substitutes what it can and leaves everything else the same:

```
f <- function() {
a <- 1
b <- 2
substitute(a + b + x)
}
f()
#> 1 + 2 + x
```

◇ To make it easier to experiment with substitute(), pryr provides the subs() function. It works exactly the same way as substitute() except it has a shorter name and if the second argument is the global environment it turns it into a list.

```
library(pryr)
#> Loading required package: Rcpp
subs(a + b + x)
#> a + b + mpg
```

◇ The second argument (to both subs() and substitute()) can override the use of the current environment, and provide an alternative list of name-value pairs to use.

```
subs(a + b, list(a = "y"))
#> "y" + b
subs(a + b, list(a = quote(y)))
#> y + b
subs(a + b, list(a = quote(y())))
#> y() + b
```

◇ Remember that every action in R is a function call, so we can also replace + with another function:

```
subs(a + b, list("+" = quote(f)))
#> f(a, b)
subs(a + b, list("+" = quote(‘*‘)))
#> a * b
```

◇ And you can use substitute to insert any arbitrary object into an expression.

◇ This is technically ok, but often results in surprising and undesirable behaviour.

```
df <- data.frame(x = 1)
(x <- subs(class(df)))
#> class(list(x = 1))
eval(x)
#> [1] "data.frame"
```

◇ Formally, substitution takes place by examining each name in the expression. If the name refers to:

- ▷ an ordinary variable, it's replaced by the value of the variable.
- ▷ a promise, it's replaced by the expression associated with the promise.
- ▷ ..., it's replaced by the contents of ... (only if the substitution occurs in a function)

◇ Otherwise the name is left as is.

```
x <- quote(mpg)
y <- quote(disp)
subs(xyplot(x ~ y, data = mtcars))
#> xyplot(mpg ~ disp, data = mtcars)
```

- ♢ It's even simpler inside a function, because we don't need to explicitly quote the x and y variables.

```
xyplot2 <- function(x, y, data = data) {
substitute(xyplot(x ~ y, data = data))
}
xyplot2(mpg, disp, data = mtcars)
#> xyplot(mpg ~ disp, data = mtcars)
```

- ♢ If we include ... in the call to substitute, we can add additional arguments to the call:

```
xyplot3 <- function(x, y, ...) {
substitute(xyplot(x ~ y, ...))
}
xyplot3(mpg, disp, data = mtcars, col = "red", aspect = "xy")
#> xyplot(mpg ~ disp, data = mtcars, col = "red", aspect = "xy")
```

- ♢ **Non-standard evaluation in substitute**
  - ▹ One application of this idea is to make a version of substitute that evaluates its first argument (i.e. a version that uses standard evaluation).
  - ▹ Note that

    ```
    x <- quote(a + b)
    substitute(x, list(a = 1, b = 2))
    #> x
    ```
  - ▹ Instead we can use pryr::substitute2:

    ```
    x <- quote(a + b)
    substitute2(x, list(a = 1, b = 2))
    #> 1 + 2
    ```
  - ▹ The implementation of substitute2 is short, but deep:

    ```
    substitute2 <- function(x, env) {
    call <- substitute(substitute(y, env), list(y = x))
    eval(call)
    }
    ```
  - ▹ It's a little tricky because of substitute()'s non-standard evaluation rules, we can't use the usual technique of working through the parentheses inside-out.
    1. First substitute(substitute(y, env), list(y = x)) is evaluated. The first argument is specially evaluated in the environment containing only one item, the value of x with the name y. Because we've put x inside a list, it will be evaluated and the rules of substitute will replace y with it's value. This yields the expression substitute(a + b, env).
    2. Next we evaluate that expression inside the current function. substitute() specially evaluates its first argument, and looks for name value pairs in env, which evaluates to list(a = 1, b = 2). Those are both values (not promises) so the result will be a + b

- ♢ **Capturing unevaluated ...**

▷ Another frequently useful technique is to capture all of the unevaluated expressions in .... Base R functions do this in many ways, but there's one technique that works well in a wide variety of situations:

```
dots <- function(...) {
eval(substitute(alist(...)))
}
```

▷ This uses the alist() function which simply captures all its arguments. This function is the same as pryr::dots(), and pryr also provides pryr::named_dots(), which ensures all arguments are named, using the deparsed expressions as default names.

The downsides of nonstandard evaluation

◇ There are usually two principles you can follow when modelling the evaluation of R code:

1. If the underlying values are the same, the results will be the same. i.e. the three results will all be the same:

   ```
   x <- 10; y <- 10
   f(10); f(x); f(y)
   ```

2. You can model evaluation by working from the innermost parentheses to the outermost.

◇ Non-standard evaluation can break both principles. This makes the mental model needed to correctly predict the output much more complicated, so it's only worthwhile to do so if there is significant gain.

◇ For example, library() and require() allow you to call them either with or without quotes, because internally they use deparse(substitute(x)) plus a couple of tricks.

◇ However, things start to get complicated if the variable has a value. For example:

```
ggplot2 <- "plyr"
library(ggplot2)
```

loads ggplot2, not plyr. If you want to load plyr (the value of the ggplot2 variable), you need to use an additional argument:

```
library(ggplot2, character.only = TRUE)
```

◇ Using an argument to change the behaviour of another argument is not a great idea because it means you must completely and carefully read all of the function arguments to understand what one function argument means. You can't understand the effect of each argument in isolation, and hence it's harder to reason about the results of a function call.

◇ There are a number of other R functions that use substitute() and deparse() in this way: ls(), rm(), data(), demo(), example(), vignette().

◇ These all use non-standard evaluation and then have special ways of enforcing the usual rules.[65]

◇ One situtation where non-standard evaluation is more useful is data.frame(), which uses the input expressions to automatically name the output variables if not otherwise provided:[66]

```
x <- 10
```

[65]Author: To me, eliminating two quotes is not worth the cognitive cost of non-standard evaluation, and I don't recommend you use substitute() for this purpose.

[66]Author: I think it is worthwhile in data.frame() because it eliminates a lot of redundancy in the common scenario when you're creating a data frame from existing variables, and importantly, it's easy to override this behaviour by supplying names for each variable.

```
y <- "a"
df <- data.frame(x, y)
names(df)
#> [1] "x" "y"
```

◇ The code for data.frame() is rather complicated, but we can create our own simple version for lists to see how a function that does this might work. The key is pryr::named_dots(), a function which returns the unevaluated ... arguments, with default names. Then it's just a matter of arranging the evaluated results in a list:

```
list2 <- function(...) {
dots <- named_dots(...)
lapply(dots, eval, parent.frame())
}
x <- 1; y <- 2
list2(x, y)
#> $x
#> [1] 1
#>
#> $y
#> [1] 2
list2(x, z = y)
#> $x
#> [1] 1
#>
#> $z
#> [1] 2
```

Applications

◇ **plyr::. and ggplot2::aes**

- ▷ ggplot2 uses the aes() to define a set of mappings between variables in your data and visual properties on your graphic.
- ▷ plyr uses the . function to capture the names (or more complicated expressions) of variables used to split a data frame into pieces.

```
. <- function (..., .env = parent.frame()) {
structure( as.list(match.call()[-1]),
env = .env,
class = "quoted"
)
}
aes <- function (x = NULL, y = NULL, ...) {
aes <- structure(
as.list(match.call()[-1]),
class = "uneval")
class(aes) <- "uneval"
ggplot2:::rename_aes(aes)
}
```

- ▷ ggplot2 provides aes_string() which allows you to specify variables by the string representation of their name, and plyr uses S3 methods so that you can either supply an object of class quoted (as created with .()), or a regular character vector.

◇ **Plyr: summarise, mutate and arrage**

- ▷ The plyr package also uses non-standard evaluation to complete the set of tools provided by the base subset() and transform() functions with mutate(), summarise() and arrange().
- ▷ Each of these functions has the same interface: the first argument is a data frame and the subsequent arguments are evaluated in the context of that data frame (i.e. they look there first for variables, and then in the current environment) and they return a data frame.
- ▷ *The code for the four function go here.
- ▷ Combined with a by operator (e.g. ddply()) these four functions allow you to express the majority of data manipulation operations.
- ▷ Then when you have a new problem, solving it becomes a matter of thinking about which operations you need to apply and in what order. The realm of possible actions has been shrunk to a manageable number.

Conclusion

◇ Now that you understand how our version of subset works, go back and read the source code for subset.data.frame, the base R version which does a little more. Other functions that work similarly are with.default, within.data.frame, transform.data.frame, and in the plyr package ., arrange, and summarise. Look at the source code for these functions and see if you can figure out how they work.

4.2 Expressions

◇ It's generally a bad idea to create code by operating on its string representation: there is no guarantee that you'll create valid code.

◇ pasting strings together will often allow you to solve your problem in the least amount of time, but it may create subtle bugs that will take your users hours to track down.

Structure of expressions

◇ We want to distinguish between the action of multiplying x by 10 and assigning the results to y versus the actual result (40). In R, we can capture the action with quote():

```
z <- quote(y <- x * 10)
z
#> y <- x * 10
```

◇ quote() gives us back an expression, an object that represents an action that can be performed by R.[67]

[67] Confusingly the expression() function produces expression lists, but since you'll never need to use that function we can safely ignore it

◇ An expression is also called an abstract syntax tree (**AST**) because it represents the abstract structure of the code in a tree form. We can use pryr::call_tree() to see the hierarchy more clearly:

```
library(pryr)
#> Loading required package: Rcpp
call_tree(z)
#> \- <-()
#> \- ‘y
#> \- *()
#> \- ‘x
#> \- 10
```

◇ There are three basic things you'll commonly see in a call tree: constants, names and calls.

▷ **constants** are atomic vectors, like "a" or 10. These appear as is.

```
call_tree(quote("a"))
#> \- "a"
call_tree(quote(1))
#> \- 1
call_tree(quote(1L))
#> \- 1L
call_tree(quote(TRUE))
#> \- TRUE
```

▷ **names** which represent the name of a variable, not its value. (Names are also sometimes called symbols). These are prefixed with '

```
call_tree(quote(x))
#> \- ‘x
call_tree(quote(mean))
#> \- ‘mean
call_tree(quote(‘an unusual name‘))
#> \- ‘an unusual name
```

▷ **calls** represent the action of calling a function, not its result. These are suffixed with (). The arguments to the function are listed below it, and can be constants, names or other calls.

```
call_tree(quote(f(a, b)))
#> \- f()
#> \- ‘a
#> \- ‘b
call_tree(quote(f(g(), h(1, a))))
#> \- f()
#> \- g()
#> \- h()
#> \- 1
#> \- ‘a
```

- As mentioned in Functions, even things that don't look like function calls still follow this same hierarchical structure[68]:

```
call_tree(quote(a + b))
#> \- +()
#> \- `a
#> \- `b
call_tree(quote(if (x > 1) x else 1/x))
#> \- if()
#> \- >()
#> \- `x
#> \- 1
#> \- `x
#> \- /()
#> \- 1
#> \- `x call_tree(quote(function(x, y) {x * y}))
#> \- function()
#> \- list(x = , y = )
#> \- {()
#> \- *()
#> \- `x
#> \- `y
#> \- structure(c(3L, 17L, 3L, 38L, 17L, 38L, 3L, 3L), srcfile = <environ...
```

- Together, names and calls are sometimes called language objects, and can be tested for with is.language(). Note that str() is somewhat inconsistent with respect to this naming convention, describing names as symbols, and calls as a language objects.

◇ **Constants**

- Quoting a single atomic vector gives it back to you:

```
is.atomic(quote(1))
#> [1] TRUE
identical(1, quote(1))
#> [1] TRUE
is.atomic(quote("test"))
#> [1] TRUE
identical("test", quote("test"))
#> [1] TRUE
```

- But quoting a vector of values gives you something different because you always use a function to create a vector:

```
identical(1:3, quote(1:3))
#> [1] FALSE
identical(c(FALSE, TRUE), quote(c(FALSE, TRUE)))
#> [1] FALSE
```

[68] In general, it's possible for any type of R object to live in a call tree, but these are the only three types you'll get from parsing R code. It's possible to put anything else inside an expression using the tools described below, but while technically correct, support is often patchy

- ▷ It's possible to use substitute() to directly insert a vector into a call tree, but use this with caution as you are creating a call that is not be generated during the normal operation of R.

```
y <- substitute(f(x), list(x = 1:3))
is.atomic(y)
#> [1] FALSE
```

♢ **Names**

- ▷ As well as capturing names with quote(), it's also possible to convert strings to names. This is mostly useful when your function receives strings as input, as it's more typing than using quote():

- ▷ As well as capturing names with quote(), it's also possible to convert strings to names. This is mostly useful when your function receives strings as input, as it's more typing than using quote():

```
identical(quote(name), as.name("name"))
#> [1] TRUE
as.name("a b")
#> `a b`
```

- ▷ Note that the second example produces the name `a b `: the backticks are the standard way of escape non-standard names in R.
- ▷ There's one special name that needs a little extra discussion: the name that represents missing values. You can get this from the formals of a function, or with alist():

```
formals(plot)$x
alist(x =)[[1]]
```

- ▷ It is basically a special name, that you can create by using quote() in a slightly unusual way:

```
quote(expr =)
```

- ▷ Note that this object behaves strangely, and is rarely useful, except when (as we'll see later) you want to try a function with arguments that don't have defaults.

```
x <- quote(expr =)
x
#> Error: argument "x" is missing, with no default
```

♢ **Calls**

- ▷ As well as capturing complete calls using quote(), modifying existing calls using substitute() you can also create calls from their constituent pieces using using as.call() or call().
- ▷ The first argument to call() is a string giving a function name, and the other arguments should be expressions that are used as the arguments to the call.

```
call(":", 1, 10)
#> 1:10
call("mean", 1:10, na.rm = TRUE)
#> mean(1:10, na.rm = TRUE)
```

- ▷ as.call() is a minor variation that takes a list where the first argument is the name of a function (not a string), and the subsequent values are the arguments.

```
x_call <- quote(1:10)
mean_call <- as.call(list(quote(mean), x_call))
identical(mean_call, quote(mean(1:10)))
#> [1] TRUE
```

▷ Note that the following two calls look the same, but are actually different:

```
(a <- call("mean", 1:10))
#> mean(1:10)
(b <- call("mean", quote(1:10)))
#> mean(1:10) identical(a, b)
#> [1] FALSE
call_tree(a)
#> \- mean()
#> \- 1:10
call_tree(b)
#> \- mean()
#> \- :()
#> \- 1
#> \- 10
```

In a, the first argument to mean is an integer vector containing the numbers 1 to 10, and in b the first argument is a call to :. You can put any R object into an expression, but the printing of expression objects will not always show the difference.

▷ The key difference is where/when evaluation occurs:

▷ You may want to capture the expression that caused the current function to run. There are two ways to do this:

```
a <- call("print", Sys.time())
b <- call("print", quote(Sys.time()))
eval(a); Sys.sleep(1); eval(a)
#> [1] "2014-01-30 19:08:18 UTC"
#> [1] "2014-01-30 19:08:18 UTC"
eval(b); Sys.sleep(1); eval(b)
#> [1] "2014-01-30 19:08:19 UTC"
#> [1] "2014-01-30 19:08:20 UTC"
```

▷ The first element of a call doesn't have to be the name of a function, and instead can be a call that generates a function:

```
(function(x) x + 1)(10)
#> [1] 11
add <- function(y) function(x) x + y
add(1)(10)
#> [1] 11
# Note that you can't create this sort of call with call
call("add(1)", 10)
#> `add(1)`(10)
# But you can with as.call
as.call(list(quote(add(1)), 10))
#> add(1)(10)
```

▷ **An interesting use of call()**

- One interesting use of call lies inside the mode<- function which is an alternative way to change the mode of a vector.

```
‘mode2<-‘ <- function (x, value) {
mde <- paste0("as.", value)
eval(call(mde, x), parent.frame())
}
x <- 1:10
mode2(x) <- "character"
x
#> [1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
```

- Another way to achieve the same goal would be find the function and then call it:

```
‘mode3<-‘ <- function(x, value) {
mde <- match.fun(paste0("as.", value))
mde(x)
}
x <- 1:10
mode3(x) <- "character"
x
#> [1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
```
[69]

▷ **Extracting elements of a call**

- When it comes to modifying calls, they behave almost exactly like lists: a call has length, '[[and [methods. The length of a call minus 1 gives the number of arguments:

```
x <- quote(read.csv("important.csv", row.names = FALSE)) length(x) - 1
#> [1] 2
```

- The first element of the call is the name of the function:

```
x[[1]]
#> read.csv
# read.csv
```

- The remaining elements are the arguments to the function, which can be extracted by name or by position.

```
x$row.names
#> [1] FALSE
x[[3]]
#> [1] FALSE
names(x)
#> [1] "" "" "row.names"
```

▷ **Standardising function calls**

- Generally, extracting arguments by position is dangerous, because R's function calling semantics are so flexible.
- It's better to match by name, but all arguments might not be named. The solution to this problem is to use match.call(), which takes a function and a call as arguments:

```
y <- match.call(read.csv, x)
names(y)
#> [1] "" "file" "row.names"
# Or if you don't know in advance what the function is
match.call(eval(x[[1]]), x)
#> read.csv(file = "important.csv", row.names = FALSE)
```

[69] Author: Generally, I'd prefer mode3<- over mode2<- because it uses concepts familiar to more R programmers, and generally it's a good idea to use the simplest and most commonly understood techniques that solve a given problem.

- This will be an important tool when we start manipulating existing function calls. If we don't use match.call we'll need a lot of extra code to deal with all the possible ways to call a function.
- We can wrap this up into a function. To figure out the definition of the associated function we evaluate the first component of the call, the name of the function.
- We need to specify an environment here, because the function might be different in different places. Whenever we provide an environment parameter, parent.frame() is usually a good default.

```
standardise_call <- function(call, env = parent.frame()) {
stopifnot(is.call(call))
f <- eval(call[[1]], env)
if (is.primitive(f)) return(call)
match.call(f, call)
}
standardise_call(y)
#> read.csv(file = "important.csv", row.names = FALSE) standardise_call(quote(standardise_call(y)))

#> standardise_call(call = y)
```

- **Modifying a call**
  - You can add, modify and delete elements of the call with the standard replacement operators, $<- and [[<-:

```
y$row.names <- TRUE
y$col.names <- FALSE
y
#> read.csv(file = "important.csv", row.names = TRUE, col.names = FALSE)
y[[2]] <- "less-important.csv"
y[[4]] <- NULL
y
#> read.csv(file = "less-important.csv", row.names = TRUE)
y$file <- quote(paste0(filename, ".csv"))
y
#> read.csv(file = paste0(filename, ".csv"), row.names = TRUE)
```

  - Calls also support the [method, but use it with care: since the first element is the function to call, removing it is unlikely to create a call that will evaluate without error.

```
x[-3] # remove the second argument
#> read.csv("important.csv")
x[-1] # remove the function name - but it's still a call!
#> "important.csv"(row.names = FALSE)
x
#> read.csv("important.csv", row.names = FALSE)
```

  - If you want to get a list of the unevaluated arguments, explicitly convert it to a list:

```
# A list of the unevaluated arguments
as.list(x[-1])
#> [[1]]
#> [1] "important.csv"
#>
#> $row.names
#> [1] FALSE
```

  - We can use these ideas to create an easy way modify a call given a list.

```
modify_call <- function(call, new_args) {
```

```
call <- standardise_call(call)
nms <- names(new_args) %||% rep("", length(new_args))
if (any(nms == "")) {
stop("All new arguments must be named", call. = FALSE)
}
for(nm in nms) {
call[[nm]] <- new_args[[nm]]
}
call
}
modify_call(quote(mean(x, na.rm = TRUE)), list(na.rm = NULL))
#> mean(x = x)
modify_call(quote(mean(x, na.rm = TRUE)), list(na.rm = FALSE))
#> mean(x = x, na.rm = FALSE)
modify_call(quote(mean(x, na.rm = TRUE)), list(x = quote(y))) #> mean(x = y, na.rm =
TRUE)
```

Parsing and deparsing

Capturing the current call

- You can convert quoted calls back and forth between text with parse() and deparse().
- deparse() takes an expression and returns a character vector. parse() does the opposite: it takes a character vector and returns a list of expressions, also known as an expression object or expression list.
- Note that because the primary use of parse() is parsing files of code on disk, the first argument is a file path, and if you have the code in a character vector, you need to use the text argument.

```
z <- quote(y <- x * 10)
deparse(z)
#> [1] "y <- x * 10"
parse(text = deparse(z))
#> expression(y <- x * 10)
```

- deparse() returns a character vector with an entry for each line, and by default it will try to make lines that are around 60 characters long. If you want a single string be sure to paste() it back together, and read the other options in the documentation.
- parse() can't return just a single expression, because there might be many top-level calls in an file.
- Instead it returns expression objects, or expression lists. You should never need to create expression objects yourself, and all you need to know about them is that they're a list of calls:

```
exp <- parse(text = c("x <- 4\ny <- x * 10"))
length(exp)
#> [1] 2
exp[[1]]
#> x <- 4
is.call(exp[[1]])
#> [1] TRUE
```

- It's not possible for parse() and deparse() be completely symmetric. See the help for deparse() for more details.
- **Sourcing files from disk**
  - With parse() and eval() you can write your own simple version of source().
  - We read in the file on disk, parse() it and then eval() each component in the specified environment.
  - This version defaults to a new environment, so it doesn't affect existing objects.
  - source() invisibly returns the result of the last expression in the file, so simple_source() does the same.[70]

```
simple_source <- function(file, envir = new.env()) {
stopifnot(file.exists(file)) stopifnot(is.environment(envir))
lines <- readLines(file, warn = FALSE)
exprs <- parse(text = lines, n = -1)
n <- length(exprs)
if (n == 0L) return(invisible())
for (i in seq_len(n - 1)) {
eval(exprs[i], envir)
}
invisible(eval(exprs[n], envir))
}
```

Capturing the current call

- You may want to capture the expression that caused the current function to run. There are two ways to do this:

  1. sys.call() captures exactly what the user typed.
  2. match.call() uses R's regular argument matching rules and converts everything to full name matching. This is usually easier to work with because you know that the call will always have the same structure.

```
f <- function(abc = 1, def = 2, ghi = 3, ...) {
list(sys = sys.call(), match = match.call())
}
f(d = 2, 2)
#> $sys
#> f(d = 2, 2)
#>
#> $match
#> f(abc = 2, def = 2)
```

- **A cautionary tale: write.csv**

[70] The real source() is considerably more complicated because it preserves the underlying source code, can echo input and output, and has many additional settings to control behaviour.

- ▷ write.csv() is a base R function where call manipulation is used in a suboptimal manner. It captures the call to write.csv() and mangles it to instead call write.table():

```
write.csv <- function (...) {
Call <- match.call(expand.dots = TRUE)
for (argname in c("append", "col.names", "sep", "dec", "qmethod")) {
if (!is.null(Call[[argname]])) {
warning(gettextf("attempt to set '%s' ignored", argname), domain = NA)
}
}
rn <- eval.parent(Call$row.names)
Call$append <- NULL
Call$col.names <- if (is.logical(rn) && !rn) TRUE else NA
Call$sep <- ","
Call$dec <- "."
Call$qmethod <- "double"
Call[[1L]] <- as.name("write.table")
eval.parent(Call)
}
```

- ▷ We could write a function that behaves identically using regular function call semantics:

```
write.csv <- function(x, file = "", sep = ",", qmethod = "double", ...) {
write.table(x = x, file = file, sep = sep, qmethod = qmethod, ...)
}
```

  - ◇ This makes the function much easier to understand: it's just calling write.table with different defaults.
  - ◇ This also fixes a subtle bug in the original write.csv: write.csv(mtcars, row = FALSE) raises an error, but write.csv(mtcars, row.names = FALSE) does not. [71]

◇ **Other uses of call capturing**

- ▷ Many modelling functions use match.call() to capture the call used to create the model.[72]
- ▷ This makes it possible to update() a model, modifying only a few components of the original model (but note that it doesn't preserve any of the computation, even if possible).
- ▷ We can rewrite it using some of the tools (dots() and modify_call()) we've developed in this chapter to make it easier to see exactly what's going on.

```
update_call <- function (object, formula., ...) {
call <- object$call
# Use a update.formula to deal with formulas like . ~ .
if (!missing(formula.)) {
call$formula <- update.formula(formula(object), formula.)
}
modify_call(call, dots(...))
}
```

[71] Generally, you always want to use the simplest tool that will solve a problem - that makes it more likely that others will understand your code. Again, there's no point in using non-standard evaluation unless there's a big win: non-standard evaluation will make your function behave much less predictably.

[72] This is one reason that creating lists of models using a function doesn't give the greatest output

```
update2 <- function(object, formula., ...) {
call <- update_call(object, formula., ...)
eval(call, parent.frame())
}
update_call(mod, formula = . ~ . + cyl)
#> lm(formula = mpg ~ wt + cyl, data = mtcars)
update_call(mod, subset = cyl == 4)
#> lm(formula = mpg ~ wt, data = mtcars, subset = cyl == 4)
```
[73]

▷ This rewrite also allows us to fix a small bug in update: it evaluates the call in the global environment, when really we want to re-evaluate it in the environment where the model was originally fit. This happens to be stored in the formula (called terms) so we can easily extract it.

```
f <- function() {
n <- 3 lm(mpg ~ poly(wt, n), data = mtcars)
}
mod <- f() update(mod, data = mtcars)
#> Error: object 'n' not found
update2 <- function(object, formula., ...) {
call <- update_call(object, formula., ...)
eval(call, environment(object$terms))
}
update2(mod, data = mtcars)
#>
#> Call:
#> lm(formula = mpg ~ poly(wt, n), data = mtcars)
#>
#> Coefficients:
#> (Intercept) poly(wt, n)1 poly(wt, n)2 poly(wt, n)3
#> 20.091 -29.116 8.636 0.275
```

▷ This is a good principle to remember: if you want to later replay the code you've captured using match.call() you really also need to capture the environment in which the code was evaluated.

▷ There is a big potential downside: because you've captured that environment and saved it in an object, that environment will hang around and any objects in the environment will also hang around. That can have big implications for memory use.

```
f <- function() {
x <- runif(1e7)
y <- runif(1e7)
lm(mpg ~ wt, data = mtcars)
}
mod <- f()
object.size(environment(mod$terms)$x)
#> 80000040 bytes
```

[73] Authoer:The original update() has an evaluate argument that controls whether the function returns a call or the result, but I think it's good principle for a function to only return one type of object (not different types depending on the arguments) so I split it into two.

Creating a function

- ♢ There's one function call that's so special it's worth devoting a little extra attention to: the function function that creates functions.
- ♢ This is one place we'll see pairlists (the object type that predated lists in R's history). The arguments of a function are stored as a pairlist: for our purposes we can treat a pairlist like a list, but we need to remember to cast arguments with as.pairlist().

```
str(quote(function(x, y = 1) x + y)[[2]])
#> Dotted pair list of 2
#> $ x: symbol
#> $ y: num 1
```

- ♢ Building up a function by hand is also useful when you can't use a closure because you don't know in advance what the arguments will be.
- ♢ The function itself is fairly simple: it creates a call to function with the args and body as arguments, and then evaluates that in the correct environment so that the function has the right scope.

```
make_function <- function(args, body, env = parent.frame()) {
args <- as.pairlist(args)
eval(call("function", args, body), env)
}[74]
add <- make_function(alist(a = 1, b = 2), quote(a + b))
add(1)
#> [1] 3
add(1, 2)
#> [1] 3[75]
```

- ♢ Note that alist() doesn't evaluate its arguments and supports arguments with and without defaults (although if you don't want a default you need to be explicit). There's one small trick if you want to have ... in the argument list: you need to use it on the left-hand side of an equals sign.

```
make_function(alist(a = , b = a), quote(a + b))
#> function (a, b = a)
#> a + b
#> <environment: 0x1e96240>
make_function(alist(a = , b = ), quote(a + b))
#> function (a, b)
#> a + b
#> <environment: 0x1e96240>
make_function(alist(a = , b = , ... =), quote(a + b))
#> function (a, b, ...)
#> a + b
#> <environment: 0x1e96240>
```

[74] pryr::make_function() includes a little more error checking but is otherwise identical.

[75] Note our use of the alist() (argument list) function. We used this earlier when capturing unevaluated ..., and we use it again here.

- ♢ If you want to mix evaluated and unevaluated arguments, it might be easier to make the list by hand:

```
x <- 1
args <- list()
args$a <- x
args$b <- quote(expr = )
make_function(args, quote(a + b))
#> function (a = 1, b)
#> a + b
#> <environment: 0x1e96240>
```

- ♢ **Unenclose**

  - ▷ Most of the time it's simpler to use closures to create new functions, but make_function() is useful if we want to make it obvious to the user what the function does (printing out a closure isn't usually that helpful because all the variables are present by name, not by value).

```
unenclose <- function(f) {
env <- environment(f)
new_body <- substitute2(body(f), env)
make_function(formals(f), new_body, parent.env(env))
}
f <- function(x) {
function(y) x + y
}
f(1)
#> function(y)
x + y
#> <environment: 0x3f89b08>
unenclose(f(1))
#> function (y)
#> 1 + y
#> <environment: 0x1e96240>
```

  - ▷ Read the documentation and source for pryr::partial() - what does it do? How does it work?

Walking the call tree with recursive functions

- ♢ The codetools package, included in the base distribution, provides some built-in tools for automated code inspection that use these ideas:

  - ▷ findGlobals(): locates all global variables used by a function. This can be useful if you want to check that your functions don't inadvertently rely on variables defined in their parent environment.
  - ▷ checkUsage(): checks for a range of common problems including unused local variables, unused parameters and use of partial argument matching.
  - ▷ The key to any function that works with the parse tree right is getting the recursion right, which means making sure that you know what the base case is (the leaves of the tree) and figuring out how to combine the results from the recursive case.

- The nodes of a tree are always calls (except in the rare case of function arguments, which are pairlists), and the leaves are names, single argument calls or constants.
- R provides a helpful function to distinguish whether an object is a node or a leaf: is.recursive().

◇ Finding F and T

- We'll start with a function that returns a single logical value, indicating whether or not a function uses the logical abbreviations T and F.[76]
- When writing a recursive function, it's useful to first think about the simplest case: how do we tell if a leaf is a T or a F?
- The base case is simple: if the object isn't recursive, then we just return the value of is_logical_abbr() applied to the object. If the object is not a node, then we work through each of the elements of the node in turn, recursively calling logical_abbr().

```
logical_abbr <- function(x) {
# Base case
if (!is.recursive(x)) return(is_logical_abbr(x))
# Recursive cases
if (is.function(x)) {
if (logical_abbr(body(x))) return(TRUE)
if (logical_abbr(formals(x))) return(TRUE)
}
else {
for (i in seq_along(x)) {
if (logical_abbr(x[[i]])) return(TRUE)
}
}
FALSE
}
logical_abbr(quote(T))
#> [1] TRUE
logical_abbr(quote(mean(x, na.rm = T)))
#> [1] TRUE
f <- function(x = TRUE) {
g(x + T)
}
logical_abbr(f)
#> [1] TRUE
```

◇ Finding all variables created by assignment

- One reason to start with this function is because the recursion is a little bit simpler - we never need to go all the way down to the leaves because we are looking for assignment, a call to <-.
- This means that our base case is simple: if we're at a leaf, we've gone too far and can immediately return.

[76] Using T and F is generally considered to be poor coding practice, and it's something that R CMD check will warn about.

- ▷ We have two other cases: we have hit a call, in which case we should check if it's <-, otherwise it's some other recursive structure and we should call the function recursively on each element.
- ▷ Note the use of identical() to compare the call to the name of the assignment function, and recall that the second element of a call object is the first argument, which for <- is the left hand side: the object being assigned to.

```
is_call_to <- function(x, name) {
is.call(x) && identical(x[[1]], as.name(name))
}
find_assign <- function(obj) {
# Base case
if (!is.recursive(obj)) return()
if (is_call_to(obj, "<-")) {
obj[[2]]
}
else {
lapply(obj, find_assign)
}
}
find_assign(quote(a <- 1))
#> a
find_assign(quote({ a <- 1 b <- 2 }))
#> [[1]]
#> NULL
#>
#> [[2]]
#> a
#>
#> [[3]]
#> b
```

- ▷ Instead of returning a list, let's keep it simple and stick with a character vector. We'll also test it with two slightly more complicated examples:

```
find_assign <- function(obj) {
# Base case
if (!is.recursive(obj)) return(character())
if (is_call_to(obj, "<-")) {
as.character(obj[[2]])
}
else {
unlist(lapply(obj, find_assign))
}
}
find_assign(quote({ a <- 1 b <- 2 a <- 3 }))
#> [1] "a" "b" "a"
find_assign(quote({ system.time(x <- print(y <- 5)) }))
#> [1] "x"
```

  ◇ This is better, but we have two problems: repeated names, and we miss assignments inside function calls.

▷ The fix for the first problem is easy: we need to wrap unique() around the recursive case to remove duplicate assignments.

```
find_assign <- function(obj) {
# Base case
if (!is.recursive(obj)) return(character())
if (is_call_to(obj, "<-")) {
call <- as.character(obj[[2]])
c(call, unlist(lapply(obj[[3]], find_assign)))
} else {
unique(unlist(lapply(obj, find_assign)))
}
}
find_assign(quote({
a <- 1
b <- 2
a <- 3
}))
#> [1] "a" "b"
find_assign(quote({
system.time(x <- print(y <- 5))
}))
#> [1] "x" "y"
```

▷ There's one more case we need to test:

```
find_assign(quote({
ls <- list()
ls$a <- 5
names(ls) <- "b"
}))
#> [1] "ls" "$" "a" "names"
call_tree(quote({
ls <- list()
ls$a <- 5
names(ls) <- "b"
}))
#> \- {()
#> \- <-()
#> \- ‘ls
#> \- list()
#> \- <-()
#> \- $()
#> \- ‘ls
#> \- ‘a
#> \- 5
#> \- <-()
#> \- names()
#> \- 'ls
#> \- "b"
```

▷ This behaviour might be ok, but we probably just want assignment into whole objects, not assignment that modifies some property of the object. Drawing the tree for that quoted object helps us see what condition we should test for - we want the object on the left hand side of assignment to be a name. This gives the final version of the find_assign function.

```
find_assign <- function(obj) {
# Base case
if (!is.recursive(obj)) return(character())
if (is_call_to(obj, "<-")) {
call <- if (is.name(obj[[2]])) as.character(obj[[2]])
c(call, unlist(lapply(obj[[3]], find_assign)))
} else {
unique(unlist(lapply(obj, find_assign)))
}
}
find_assign(quote({
ls <- list()
ls$a <- 5
names(ls) <- "b"
}))
#> [1] "ls"
```
[77]

◇ **Modifying the call tree**

▷ Instead of returning vectors computed from the contents of an expression, you can also return a modified expression, such as base R's bquote().

▷ bquote() is a slightly more flexible form of quote: it allows you to optionally quote and unquote some parts of an expression (it's similar to the backtick operator in Lisp).

▷ Everything is quoted, unless it's encapsulated in .() in which case it's evaluated and the result is inserted.

```
a <- 1 b <- 3 bquote(a + b)
#> a + b
bquote(a + .(b))
#> a + 3
bquote(.(a) + .(b))
#> 1 + 3
bquote(.(a + b))
#> [1] 4
```

[77] Making this function work absolutely correct requires quite a lot more work, because we need to figure out all the other ways that assignment might happen: with =, assign(), or delayedAssign(). But a static tool can never be perfect: the best you can hope for is a set of heuristics that catches the most common 90% of cases.

▷ This provides a fairly easy way to control what gets evaluated when you call bquote(), and what gets evaluated when the expression is evaluated.

```
bquote2 <- function (x, where = parent.frame()) {
# Base case
if (!is.recursive(x)) return(x)
if (is.call(x)) {
if (identical(x[[1]], quote(.))) {
# Call to .(), so evaluate
eval(x[[2]], where)
} else {
as.call(lapply(x, bquote2, where = where))
}
} else if (is.pairlist(x)) {
as.pairlist(lapply(x, bquote2, where = where))
} else {
stop("Unknown case")
}
}
x <- 1
bquote2(quote(x == .(x)))
#> x == 1
y <- 2 bquote2(quote(function(x = .(x)) {
x + .(y)
}))
#> function(x = 1) {
#> x + 2
#> }
```

▷ Note that functions that modify the source tree are most useful for creating expressions that are used at run-time, not saved back into the original source file. That's because all non-code information is lost:

```
bquote2(quote(function(x = .(x)) {
# This is a comment
x + # funky spacing
.(y)
}))
#> function(x = 1) {
#> x + 2
#> }
```

▷ bquote() is rather like a macro from a languages like Lisp. But unlike macros the modifications occur at runtime, not compile time (which doesn't have any meaning in R). And unlike a macro there is no restriction to return an expression: a macro-like function in R can return anything. More like fexprs. a fexpr is like a function where the arguments aren't evaluated by default; or a macro where the result is a value, not code.

▷ Programmer's Niche: Macros in R by Thomas Lumley. http://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf#page=11

4.3 Special environments

◇ R's lexical scoping rules, lazy argument evaluation and first-class environments make it an excellent language in which to design special environments that allow you to create **domain specific languages (DSLs)**.

◇ This chapter uses ideas from the breadth of the book including non-standard evaluation, environment manipulation, active bindings, ...

- ▷ Evaluate code in a special context: local, capture.output, with_*
- ▷ Supply an expression instead of a function: curve
- ▷ Combine evaluation with extra processing: test_that, assert_that
- ▷ Create a full-blown DSL: html, plotmath, deriv, parseNamespace, sql

Evaluation code in a special context

◇ It's often useful to evaluate a chun of code in a special context

- ▷ Temporarily modifying global state, like the working directory, environment variables, plot options or locale.
- ▷ Capture side-effects of a function, like the text it prints, or the warings it emits
- ▷ Evaluate code in a new environmentw

◇ with_something

- ▷ There are a number of parameters in R that have global state (e.g. option(), environmental variables, par(), ...) and it's useful to be able to run code temporarily in a different context.
- ▷ it's often useful to be able to run code with the working directory temporarily set to a new location:

```
in_dir <- function(dir, code) {
old <- setwd(dir)
on.exit(setwd(old))
force(code)
}
getwd()
#> [1] "/home/travis/build/hadley/adv-r"
in_dir("~", getwd())
#> [1] "/home/travis"
```

The basic pattern is simple:

1. We first set the directory to a new location, capturing the current location from the output of setwd.
2. We then use on.exit() to ensure that the working directory is returned to the previous value regardless of how the function exits.
3. Finally, we explicitly force evaluation of the code. (We don't actually need force() here, but it makes it clear to readers what we're doing)

- ▷ We could use similar code to temporarily set global options:

```
with_options <- function(opts, code) {
old <- options(opts)
on.exit(options(old))
force(code)
}
x <- 0.123456
print(x)
#> [1] 0.1235
with_options(list(digits = 3), print(x))
#> [1] 0.123
with_options(list(digits = 1), print(x))
#> [1] 0.1
```

- ▷ We could extract out that commonality with a function operator that takes the setter function as an input:

```
with_something <- function(set) {
function(new, code) {
old <- set(new)
on.exit(set(old))
force(code)
}
}
```

  - ◇ Then we can easily generate a whole set of with functions:

```
in_dir <- with_something(setwd)
with_options <- with_something(options)
with_par <- with_something(par)
```

- ▷ However, many of the setter functions that affect global state in R don't return the previous values in a way that can easily be passed back in.
- ▷ In that case, like for .libPaths(), which controls where R looks for packages to load, we first create a wrapper that enforces the behaviour we want, and then use with_something():

```
set_libpaths <- function(paths) {
libpath <- normalizePath(paths, mustWork = TRUE)
old <- .libPaths()
.libPaths(paths)
invisible(old)
}
with_libpaths <- with_something(set_libpaths)
```

♢ capture.output

- ▷ capture.output() is a useful function when the output you really want from a function is printed to the console.

```
y <- 1:10
y_str <- str(y)
#> int [1:10] 1 2 3 4 5 6 7 8 9 10
y_str
#> NULL
y_str <- capture.output(str(y))
y_str
#> [1] " int [1:10] 1 2 3 4 5 6 7 8 9 10"
```

- ▷ To work its magic, capture.output() uses sink(), which allows you to redirect the output stream to an arbitrary connection.
  - ◇ first write a helper function that allows us to execute code in the context of a sink(), automatically un-sink()ing when the function finishes:
    ```
    with_sink <- function(connection, code, ...) {
    sink(connection, ...)
    on.exit(sink())
    code
    }
    with_sink("temp.txt", print("Hello"))
    readLines("temp.txt")
    #> [1] "[1] \"Hello\""
    file.remove("temp.txt")
    #> [1] TRUE
    ```
  - ◇ add a little extra wrapping to our capture.output2() to write to a temporary file, read from it and clean up after ourselves:
    ```
    capture.output2 <- function(code) {
    temp <- tempfile()
    on.exit(file.remove(temp))
    with_sink(temp, force(code))
    readLines(temp)
    }
    capture.output2(cat("a", "b", "c", sep = "\n"))
    #> [1] "a" "b" "c"
    ```
  - ◇ The real capture.output() is a bit more complicated: it uses a local textConnection to capture the data sent to sink, and it allows you to supply multiple expressions which are evaluated in turn. Using with_sink() this looks like capture.output3()
    ```
    capture.output3 <- function(..., env = parent.frame()) {
    txtcon <- textConnection("rval", "w", local = TRUE)
    with_sink(txtcon, {
    args <- dots(...)
    for(i in seq_along(args)) {
    out <- withVisible(eval(args[[i]], env))
    if (out$visible) print(out$value)
    }
    })
    rval
    }
    ```
  - ◇ If you want to capture more types of output (like messages and warnings), you may find the evaluate package helpful. It powers knitr, and does its best to ensure high fidelity between its output and what you'd see if you copied and pasted the code at the console.

◇ Evaluating code in a new environment

- ▷ In the process of performing a data analysis, you may create variables that are necessarily because they help break a complicated sequence of steps down in to easily digestible chunks, but are not needed afterwards.
- ▷ It's useful to be able to store only the final result, preventing the intermediate results from cluttering your workspace. We already know one way of doing this, using a function[78]:

[78]In JavaScript this is called the immediately invoked function expression (IIFE), and is used extensively in modern JavaScript

```
x <- (function() {
a <- 10 b <- 30 a + b
})()
```

▷ R provides another tool that's a little less verbose, the local() function:

```
x <- local({
a <- 10 b <- 30 a + b
})
```

▷ The idea of local is to create a new environment (inheriting from the current environment) and run the code in that. The essence of local() is captured in this code:

```
local2 <- function(expr) {
envir <- new.env(parent = parent.frame())
eval(substitute(expr), envir)
}
```

▷ The real local() code is considerably more complicated because it adds a second environment parameter.[79]

▷ The original code is also hard to understand because it is very concise and uses some sutble features of evaluation (including non-standard evaluation of both arguments). If you have read metaprogramming, you might be able to puzzle it out, but to make it a bit easier I have rewritten it in a simpler style below.

```
local2 <- function(expr, envir = new.env()) {
env <- parent.frame()
call <- substitute(eval(quote(expr), envir))
eval(call, env)
}
a <- 100
local2({
b <- a + sample(10, 1)
my_get <<- function() b
})
my_get()
#> [1] 103
```

▷ You might wonder we can't simplify to this:

```
local3 <- function(expr, envir = new.env()) {
eval(substitute(expr), envir)
}
```

But it's because of how the arguments are evaluated - default arguments are evalauted in the scope of the function so that local(x) would not be the same as local(x, new.env()) without special effort.

to encapsulate different JavaScript libraries

[79] I don't think this is necessary because if you have an explicit environment parameter, then you can already evaluate code in that environment with evalq().

Anaphoric functions

- Another variant along these lines is an "**anaphoric function**", or a function that uses a pronoun.
- Example: curve() draws a plot of the specified function, but interestingly you don't need to use a function, you just supply an expression that uses x:

  ```
  curve(x ^ 2)
  curve(sin(x), to = 3 * pi)
  curve(sin(exp(4 * x)), n = 1000)
  ```

- Here x plays a role like a pronoun in an English sentence: it doesn't represent a single concrete value, but instead is a place holder that varies over the range of the plot.
- Note that it doesn't matter what the value of x outside of curve() is: the expression is evaluated in a special environment where x has a special meaning:

  ```
  x <- 1 curve(sin(exp(4 * x)), n = 1000)
  ```

- The essence of curve(), omitting many useful but incidental details like plot labelling, looks like this:

  ```
  curve2 <- function(expr, xlim = c(0, 1), n = 100, env = parent.frame()) {
  env2 <- new.env(parent = env)
  env2$x <- seq(xlim[1], xlim[2], length = n)
  y <- eval(substitute(expr), env2)
  plot(env2$x, y, type = "l", ylab = deparse(substitute(expr)))
  }
  curve2(sin(exp(4 * x)), n = 1000)
  ```

- Creating a new environment containing the pronoun is the key technique for implementing anaphoric functions.
- Another way to solve the problem would be to turn the expression into a function using make_function():

  ```
  curve3 <- function(expr, xlim = c(0, 1), n = 100, env = parent.frame()) {
  f <- pryr::make_function(alist(x = ), substitute(expr), env)
  x <- seq(xlim[1], xlim[2], length = n)
  y <- f(x)
  plot(x, y, type = "l", ylab = deparse(substitute(expr)))
  }
  curve3(sin(exp(4 * x)), n = 1000)[80]
  ```

- All anaphoric functions need careful documentation so that the user knows that some variable will have special properties inside the anaphoric function and must otherwise be avoided.
- If you're interesting in learning more, there are some good resources for anaphoric functions in Arc (a list like language)http://www.arcfn.com/doc/anaphoric.html, Perlhttp://www.perlmonks.org/index.pl?node_id=666047 and Clojurehttp://amalloy.hubpages.com/hub/Unhygenic-anaphoric-Clojure-macros

[80] Author: I would have a slight preference for the second because it would be easier to reuse the part of the curve3() that turns an expression into a function.

4.4 Domain specific languages

- ◇ Embedded DSLs take advantage of a host language's parsing and execution framework, but adjust the semantics somewhat to make them more suitable for a specific task.
- ◇ R already has a simple and popular DSL built in: the formula specification, which offers a succinct way of describing the relationship between predictors and the response.
- ◇ Another package that makes extensive use of these ideas is dplyr, which provides `translate_sql()` to convert R expressions into SQL:

```
library(dplyr) translate_sql(sin(x) + tan(y))
#> <SQL> SIN("x") + TAN("y")
translate_sql(x < 5 & !(y >= 5))
#> <SQL> "x" < 5.0 AND NOT(("y" >= 5.0))
translate_sql(first %like% "Had*")
#> <SQL> "first" LIKE 'Had*'
translate_sql(first %in% c("John", "Roger", "Robert"))
#> <SQL> "first" IN ('John', 'Roger', 'Robert')
translate_sql(like == 7)
#> <SQL> "like" = 7.0
```

- ◇ An important part of the overall structure of the package is `partial_eval()` which helps manage expressions where some of the components refer to variables in the database and some refer to local R objects.
- ◇ R is well suited for hosting DSLs because the combination of a small amount of computing on the language and constructing special evaluation environments is very powerful.
- ◇ If you're interested in learning more, I highly recommend *Domain Specific Languages* by Martin Fowler: it discusses many options for creating a DSL and provides many examples of different languages.

HTML

- ◇ HTML is the language that underlies the majority of the web. It is a special case of SGML, and similar (but not identical) to XML.
- ◇ HTML is composed of tags that look like `<tag></tag>`.
- ◇ Tags can be contained inside other tags and intermingled with text.
- ◇ Generally, HTML ignores whitespace: an sequence of whitespace is equivalent to a single space.
- ◇ There are over 100 HTML tags, but to illustrate HTML we're going to focus on just a few:
  - ▷ `<body>`: the top-level tag that all content is enclosed within
  - ▷ `<h1>`: creates a heading-1, the top level heading
  - ▷ `<p>`: creates a paragraph
  - ▷ `<b>`: emboldens text
  - ▷ `<img>`: embeds an image

◇ Two important attributes used on just about every tag are id and class. These are used in conjunction with CSS (cascading style sheets) in order to control the style of the document.

◇ Some tags, like <img>, can't have any content. These are called void tags and have a slightly different syntax: instead of writing <img></img> you write <img />. Since they have no content, attributes are more imporant, and img has three that are used for almost every image: src (where the image lives), width and height.

◇ Because < and > have special meanings in HTML, you can't write them directly. Instead you have to use the HTML escapes &gt; and &lt;. And since those escapes use &, you also have to escape it with &amp; if you want a literal ampersand.

◇ **Goal**

- ▷ Our goal is to make it easy to generate HTML from R. To give a concrete example, we want to generate the following HTML:

```
with_html(body(
h1("A heading", id = "first"),
p("Some text &", b("some bold text.")),
img(src = "myimg.png", width = 100, height = 100)
))
```

Note that the nesting of function calls is the same as the nesting of tags, unnamed arguments become the content of the tag, and named arguments become the attributes.

◇ **Escaping**

- ▷ The easiest way to do this is to create an S3 class that allows us to distinguish between regular text (that needs escaping) and HTML (that doesn't).
- ▷ We then write an escape method that leaves HTML unchanged and escapes the special characters (&, <, >) in ordinary text. We also add a method for lists for convenience

```
html <- function(x) structure(x, class = "html")
print.html <- function(x, ...) cat("<HTML> ", x, "\n", sep = "")
escape <- function(x) UseMethod("escape")
escape.html <- function(x) x
escape.character <- function(x) {
    x <- gsub("&", "&amp;", x)
    x <- gsub("<", "&lt;", x)
    x <- gsub(">", "&gt;", x)
    html(x)
}
escape.list <- function(x) {
    lapply(x, escape.character)
}
# Now we check that it works
escape("This is some text.")

## <HTML> This is some text.

escape("x > 1 & y < 2")

## <HTML> x &gt; 1 &amp; y &lt; 2
```

```r
# Double escaping is not a problem
escape(escape("This is some text. 1 > 2"))

## <HTML> This is some text. 1 &gt; 2

# And text we know is HTML doesn't get escaped.
escape(html("<hr />"))

## <HTML> <hr />
```

◇ **Basic tag functions**

▷ HTML tags can have both attributes (e.g. id, or class) and children (like <b> or <i>).

▷ We need some way of separating these in the function call: since attributes are named values and children don't have names, it seems natural to separate using named vs. unnamed arguments.

▷ Then a call to p() might look like:

```r
p("Some text.", b("some bold text"), class = "mypara")
```

▷ We could list all the possible attributes of the p tag in the function definition, but that's hard because there are so many, and it's possible to use custom attributes Instead we'll just use ... and separate the components based on whether or not they are named.

▷ To do this correctly, we need to be aware of a "feature" of names():

```r
names(c(a = 1, b = 2))

## [1] "a" "b"

names(c(a = 1, 2))

## [1] "a" ""

names(c(1, 2))

## NULL
```

▷ With this in mind we create two helper functions to extract the named and unnamed components of a vector:

```r
named <- function(x) {
    if (is.null(names(x)))
        return(NULL)
    x[names(x) != ""]
}

unnamed <- function(x) {
    if (is.null(names(x)))
        return(x)
    x[names(x) == ""]
}
```

▷ We can now create our p() function. There's one new function here: html_attributes().

```
source("code/html-attributes.r", local = TRUE)
p <- function(...) {
args <- list(...)
attribs <- html_attributes(named(args))
children <- unlist(escape(unnamed(args)))

html(paste0(
"<p", attribs, ">",
paste(children, collapse = ""),
"</p>"
 ))
}
p("Some text")
#> [1] "<p>Some text</p>"
#> attr(,"class")
#> [1] "html"
p("Some text", id = "myid")
#> [1] "<p id = 'myid'>Some text</p>"
#> attr(,"class")
#> [1] "html"
p("Some text", image = NULL)
#> [1] "<p image>Some text</p>"
#> attr(,"class")
#> [1] "html"
p("Some text", class = "important", "data-value" = 10)
#> [1] "<p class = 'important' data-value = '10'>Some text</p>"
#> attr(,"class")
#> [1] "html"
```

◊ **Tag functions**

▷ With this definition of p() it's pretty easy to see what will change for different tags: we just need to replace "p" with a variable. [81]

```
tag <- function(tag) {
force(tag)
function(...) {
 args <- list(...)
 attribs <- html_attributes(named(args))
 children <- unlist(escape(unnamed(args)))
html(paste0(
"<", tag, attribs, ">",
paste(children, collapse = ""),
"</", tag, ">"
))
}
}
```

▷ Before we continue to generate functions for every possible HTML tag, we need a variant of `tag()` for void tags. It can be very similar to `tag()`, but needs to throw an error if there are any unnamed

[81] We're forcing the evaluation tag with the expectation we'll be calling this function from a loop later on - that avoids potential bugs caused by lazy evaluation.

tags, and the tag itself looks slightly different:

▷
```
void_tag <- function(tag) {
force(tag)
function(...) {
args <- list(...)
if (length(unnamed(args)) > 0) {
stop("Tag ", tag, " can not have children", call. = FALSE)
}
attribs <- html_attributes(named(args))          html(paste0("<", tag, attribs, " />"))
} }
img <- void_tag("img")
img(src = "myimage.png", width = 100, height = 100)
#> [1] "<img src = 'myimage.png' width = '100' height = '100' />"
#> attr(,"class")
#> [1] "html"
```

♢ Processing all tags

- ▷ Next we need a list of all the HTML tags:

LaTeX

♢ LaTeX mathematics are complex, and well documented. They have a fairly simple structure:

- ▷ Most simple mathematical equations are represented in the way you'd type them into R: x * y, z ^ 5. Subscripts are written using _, e.g. x_1.
- ▷ Special characters start with a \: \pi = π, \pm = ±, and so on. There are a huge number of symbols available in LaTeX. Googling for latex math symbols finds many lists, and there's even a service where you can sketch a symbol in the browser and it will look it up for you.
- ▷ More complicated functions look like \name{arg1}{arg2}. For example to represent a fraction you use \frac{a}{b}, and a sqrt looks like \sqrt{a}.
- ▷ To group elements together use {}: i.e. x ^ a + b vs. x ^ {a + b}
- ▷ In good math typesetting, a distinction is made between variables and functions, but without extra information, LaTeX doesn't know whether f(a * b) represents calling the function f with argument a * b, or is shorthand for f * a * b. If f is a function, you can tell LaTeX to typeset it using an upright font with `\textrm{f}(a * b)`

♢ **Goal**

- ▷ Our goal is to use these rules to automatically convert from an R expression to a LaTeX representation of that expression. We will tackle it in four stages:
  1. Convert known symbols: pi -> \pi
  2. Leave other symbols unchanged: x -> x, y -> y
  3. Convert known functions: x * pi -> x * \pi, sqrt(frac(a, b)) -> \sqrt{\frac{a, b}}
  4. Wrap unknown functions with \textrm: f(a) -> \textrm{f}(a)
- ▷ **to_math**
  - ⋄ To begin, we need a wrapper function that we'll use to convert R expressions into LaTeX math expressions.

◇ This works the same way as to_html: we capture the unevaluated expression and evaluate it in a special environment.

◇ However, the special environment is no longer fixed, and will vary depending on the expression.

```
to_math <- function(x) {
expr <- substitute(x)
eval(expr, latex_env(expr))
}
```

◇ **Known symbols**

▷ Our first step is to create an environment that allows us to convert the special LaTeX symbols used for Greek, e.g. pi to \pi.

▷ First we create than environment by creating a named vector, converting that vector into a list, and then turn that list into an environment.

```
greek <- c("alpha", "theta", "tau", "beta", "vartheta", "pi", "upsilon", "gamma",
    "gamma", "varpi", "phi", "delta", "kappa", "rho", "varphi", "epsilon", "lambda",
    "varrho", "chi", "varepsilon", "mu", "sigma", "psi", "zeta", "nu", "varsigma",
    "omega", "eta", "xi", "Gamma", "Lambda", "Sigma", "Psi", "Delta", "Xi",
    "Upsilon", "Omega", "Theta", "Pi", "Phi")
greek_list <- setNames(paste0("\\", greek), greek)
greek_env <- list2env(as.list(greek_list), parent = emptyenv())
```

▷ Now we can check it

```
to_math <- function(x) {
    expr <- substitute(x)
    eval(expr, latex_env(expr))
}

latex_env <- function(expr) {
    greek_env
}

to_math(pi)

## [1] "\\pi"

to_math(beta)

## [1] "\\beta"
```

◇ **Unknown symbols**

▷ If a symbol isn't greek, we want to leave it as is. This is trickier because we don't know in advance what symbols will be used, and we can't possibly generate them all.

▷ So we'll use a little bit of computing on the language to find out what symbols are present in an expression.

▷ The all_names function takes an expression: if it's a name, it converts it to a string; if it's a call, it recurses down through its arguments.

```
all_names <- function(x) {
    # Base cases
    if (is.name(x))
        return(as.character(x))
    if (!is.call(x))
        return(NULL)

    # Recursive case
    children <- lapply(x[-1], all_names)
    unique(unlist(children))
}
all_names(quote(x + y + f(a, b, c, 10)))

## [1] "x" "y" "a" "b" "c"
```

▷ We now want to take that list of symbols, and convert it to an environment so that each symbol is mapped to a string representing itself (e.g. so `eval(quote(x), env)` yields `"x"`). We again use the pattern of converting a named character vector to a list, then an environment.

```
latex_env <- function(expr) {
    names <- all_names(expr)
    symbol_list <- setNames(as.list(names), names)
    symbol_env <- list2env(symbol_list)

    symbol_env
}
to_math(x)

## [1] "x"

to_math(longvariablename)

## [1] "longvariablename"

to_math(pi)

## [1] "pi"
```

▷ This works, but we need to combine it with the enviroment of the Greek symbols. Since we want to prefer Greek to the defaults (e.g. `to_math(pi)` should give `"\\pi"`, not `"pi"`), `symbol_env` needs to be the parent of `greek_env`, and thus we need to make a copy of greek_env with a new parent.

▷ Strangely R doesn't come with a function for cloning environments, but we can easily create one by combining two existing functions:

```
clone_env <- function(env, parent = parent.env(env)) {
    list2env(as.list(env), parent = parent)
}
```

▷ This gives us a function that can convert both known (Greek) and unknown symbols.

```
latex_env <- function(expr) {
    # Unknown symbols
    names <- all_names(expr)
    symbol_list <- setNames(as.list(names), names)
    symbol_env <- list2env(symbol_list)

    # Known symbols
    clone_env(greek_env, symbol_env)
}

to_math(x)

## [1] "x"

to_math(longvariablename)

## [1] "longvariablename"

to_math(pi)

## [1] "\\pi"
```

◇ **Known functions**

▷ We'll start with a couple of helper closures that make it easy to add new unary and binary operators. These functions are very simple since they only have to assemble strings.

```
unary_op <- function(left, right) {
    force(left)
    force(right)
    function(e1) {
        paste0(left, e1, right)
    }
}

binary_op <- function(sep) {
    force(sep)
    function(e1, e2) {
        paste0(e1, sep, e2)
    }
}
```

▷ Using these helpers, we can map a few illustrative examples from R to LaTeX.

```
# Binary operators
f_env <- new.env(parent = emptyenv())
f_env$"+" <- binary_op(" + ")
f_env$"-" <- binary_op(" - ")
f_env$"^" <- binary_op("^")

# Grouping
f_env$"{" <- unary_op("\\left{ ", " \\right}")
f_env$"(" <- unary_op("\\left( ", " \\right)")
f_env$paste <- paste
```

```
# Other math functions
f_env$sqrt <- unary_op("\\sqrt{", "}")
f_env$sin <- unary_op("\\sin(", ")")
f_env$log <- unary_op("\\log(", ")")
f_env$frac <- function(a, b) {
    paste0("\\frac{", a, "}{", b, "}")
}

# Labelling
f_env$hat <- unary_op("\\hat{", "}")
f_env$tilde <- unary_op("\\tilde{", "}")
```

▷ We again modify latex_env() to include this environment. It should be the last environment in which names are looked for, so that sin(sin) works.

```
latex_env <- function(expr) {
    # Known functions
    f_env

    # Default symbols
    names <- all_names(expr)
    symbol_list <- setNames(as.list(names), names)
    symbol_env <- list2env(symbol_list, parent = f_env)

    # Known symbols
    greek_env <- clone_env(greek_env, parent = symbol_env)
}
to_math(sin(x + pi))

## [1] "\\sin(x + \\pi)"

to_math(log(x_i^2))

## [1] "\\log(x_i^2)"

to_math(sin(sin))

## [1] "\\sin(sin)"
```

◇ Unknown functions

▷ Finally, we'll add a default for functions that we don't know about. Like the unknown names, we can't know in advance what these will be, so we again use a little computing on the language to figure them out:

```
all_calls <- function(x) {
    # Base name
    if (!is.call(x))
        return(NULL)

    # Recursive case
    fname <- as.character(x[[1]])
    children <- lapply(x[-1], all_calls)
    unique(c(fname, unlist(children, use.names = FALSE)))
}

all_calls(quote(f(g + b, c, d(a))))

## [1] "f" "+" "d"
```

▷ And we need a closure that will generate the functions for each unknown call

```
unknown_op <- function(op) {
    force(op)
    function(...) {
        contents <- paste(..., collapse = ", ")
        paste0("\\mathrm{", op, "}(", contents, ")")
    }
}
```

▷ And again we update latex_env():

```
latex_env <- function(expr) {
    calls <- all_calls(expr)
    call_list <- setNames(lapply(calls, unknown_op), calls)
    call_env <- list2env(call_list)

    # Known functions
    f_env <- clone_env(f_env, call_env)

    # Default symbols
    symbols <- all_names(expr)
    symbol_list <- setNames(as.list(symbols), symbols)
    symbol_env <- list2env(symbol_list, parent = f_env)

    # Known symbols
    greek_env <- clone_env(greek_env, parent = symbol_env)
}
to_math(f(a * b))

## [1] "\\mathrm{f}(\\mathrm{*}(a b))"
```

5 Performant Code

5.1 Performance

Introduction

◇ R is not a fast computer language. This is not an accident: R has been thoughtfully designed to make it easier for you to solve data analysis and statistics challenges, not to make your computer's life easier.

Why is R slow?

◇ To understand R's performance it helps to think about R in two ways: as a language and an implementation.

◇ The R-language is abstract. It defines what R code means and how it should work.

◇ An implementation is concrete: you give it R code and it computes the result.

◇ R has the R language definition, but it is informal and incomplete: the R-language is mostly defined in terms of how GNU-R works.

◇ Even though the distinction between R-language and GNU-R isn't clear cut, it's still useful because poor performance due to the implementation can be fixed relatively easily, while poor performance related to the language is hard to fix without changing what R code means.

◇ While language design constrains the maximum possible performance, GNU-R is currently far from the optimum.

◇ As well as performance limitations imposed by the language design and current implementation, a lot of R code is slow because it's poorly written.

Microbenchmarking

◇ A **microbenchmark** is a performance measurement of a very small piece of code, something that might take microseconds (µs) or nanoseconds (ns) to run.

◇ the small times in microbenchmarks are typically dominated by higher-order effects in real code. Don't change the way you code because of these microbenchmarks; instead wait until the next chapter to see how to improve the performance of real code.

◇ The best tool for microbenchmarking in R is the **microbenchmark** package. It provides very precise timings and makes it possible to compare operations that only take a tiny amount of time.

```
library(microbenchmark)
x <- runif(100)
microbenchmark(sqrt(x), x^0.5)

## Unit: microseconds
##     expr   min    lq median    uq    max neval
##  sqrt(x) 1.189 1.236  1.281 1.349 22.642   100
##    x^0.5 1.394 1.440  1.484 1.567  6.923   100
```

◇ To help calibrate the impact of a microbenchmark on run time, it's useful to think about how many times a function needs to run before it takes a second. If a microbenchmark takes:

- ▷ 1 ms, then one thousand calls takes a second

  - ▷ 1 µs, then one million calls takes a second
  - ▷ 1 ns, then one billion calls takes a second

- ◇ microbenchmark() takes multiple expressions as input, and displays summaries of the distribution of times.

Language performance

- ◇ You can't benchmark a language, since it's an abstract construct, so the benchmarks are only suggestive of the cost of these decisions to the language, but are nevertheless useful.
- ◇ Designing a useful language is a delicate balancing act. There are many options and many tradeoffs, and you need to balance between speed, flexibility and ease of implementation.
- ◇ If you'd like to learn more about the performance characteristics of the R-language and how they affect real code, I highly recommend *Evaluating the Design of the R Language*[82] by Floreal Morandat, Brandon Hill, Leo Osvald and Jan Vitek.
- ◇ **Extreme dynamism**
  - ▷ R is an extremely dynamic programming language, and almost anything can be modified after it is created.
    - ◇ change the body, arguments and environment of functions
    - ◇ change the S4 methods for a generic
    - ◇ add new fields to an S3 object, or even change its class
    - ◇ modify objects outside of the local environment with <<-
  - ▷ Pretty much the only thing you can't change are objects in sealed namespaces. After a package is loaded its namespace is sealed and it is harder (although still not impossible) to change objects defined by the package.
  - ▷ The advantage of dynamism is that you don't need to do upfront planning, and you don't need an initial compilation step. You can change your mind at any point, iterating you way to a solution without having to start afresh.
  - ▷ The disadvantage of dynamism is that it makes it difficult to predict exactly what will happen for a given function call.
  - ▷ If an interpreter can't predict what's going to happen, it has to look through many options to find the right one, a slow operation.
  - ▷ For example, code like the following loop is slow in R, because R can't know that the type of x never changes. That means R has to look for the right + method (e.g. is it adding doubles, or integers) in every iteration of the loop. The cost of finding the right method is higher for non-primitive functions.

```r
x <- 0L
for (i in 1:1e+06) {
    x <- x + 1
}
```

  - ▷ The following microbenchmark illustrates the cost of method dispatch for S3, S4, and RC.

[82] *https://www.cs.purdue.edu/homes/jv/pubs/ecoop12.pdf* It discusses a powerful methodology for understanding the performance characteristics of GNU-R using a modified R interpreter and a wide set of code found in the wild.

```
f <- function(x) NULL
s3 <- function(x) UseMethod("s3")
s3.integer <- f

A <- setClass("A", representation(a = "list"))
setGeneric("s4", function(x) standardGeneric("s4"))
setMethod(s4, "A", f)

B <- setRefClass("B", methods = list(rc = f))

a <- A()
b <- B$new()

microbenchmark(
fun = f(),
S3 = s3(1L),
S4 = s4(a),
RC = b$rc()
)
#> Unit: nanoseconds
#>  expr    min     lq median     uq       max neval
#>   fun    432    595    662    762     1,340   100
#>    S3  5,390  6,000  6,550  6,820    40,300   100
#>    S4 26,700 28,100 28,900 29,700   109,000   100
#>    RC 30,000 31,300 32,100 33,000 1,150,000   100
```

▷ the function call takes about 280 ns. S3 method dispatch takes an additional 3,000 ns; S4 dispatch, 13,000 ns; and RC dispatch, 11,000 ns. S3 and S4 method dispatch is so expensive because R must search for the right method every time the generic is called; it might have changed between this call and the last.

▷ R could do better by caching methods between calls, but caching is hard to do correctly and a notorious source of bugs.

◇ **Name lookup with mutable environments**

▷ It's surprisingly difficult to find the value associated with a name in the R-language because of the combination of lexical scoping and extreme dynamism.

▷ Take the following example. Each time we print a it comes from a different environment:

```
a <- 1
f <- function() {
    g <- function() {
        print(a)
        assign("a", 2, envir = parent.frame())
        print(a)
        a <- 3
        print(a)
    }
    g()
}
f()
```

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

▷ This means that you can't rely on the binding associated with a name being in the same place as it was last time you looked: you have to start from scratch each time.

▷ Since many basic functions are in the base environment, R has to look through every environment on the search path, which could easily be 10 or 20 environments.

▷ The following microbenchmark hints at the performance costs. We create four versions of f(), each with one more environment (containing 26 bindings) between the environment of f() and the base environment where +, ^, (, and { are defined.

```
random_env <- function(parent = globalenv()) {
letter_list <- setNames(as.list(runif(26)), LETTERS)
list2env(letter_list, envir = new.env(parent = parent))
}

set_env <- function(f, e) {
environment(f) <- e
f
}
f2 <- set_env(f, random_env())
f3 <- set_env(f, random_env(environment(f2)))
f4 <- set_env(f, random_env(environment(f3)))

microbenchmark(
f(1, 2),
f2(1, 2),
f3(1, 2),
f4(1, 2),
times = 1000
)
#> Unit: nanoseconds
#>      expr   min    lq median    uq    max neval
#>   f(1, 2) 2,500 3,250  3,370 3,520 45,400  1000
#>  f2(1, 2) 1,770 2,430  2,540 2,680 14,800  1000
#>  f3(1, 2) 1,930 2,550  2,660 2,770 10,600  1000
#>  f4(1, 2) 2,030 2,690  2,810 2,940 11,900  1000
```

Each additional environment between f() and the base environment makes the function slower by about 50ns.

▷ It might be possible to implement a caching system so that R only needs to look up the value of each name once. This is hard because there are so many ways to change the value associated with a name: <<-, assign(), eval(), ...

▷ Any caching system would have to connect to each of these tools to make sure the cache was correctly invalidated and you didn't get an old (incorrect) value.

▷ Another simple fix would be to add more built in constants that you can't override. This, for example, would mean that you always know exactly what +, -, { and (mean, and you don't need to repeatedly look up their definitions.

◇ **Lazy evaluation overhead**

- ▷ In R, functions arguments are evaluated lazily (as discussed in lazy evaluation and capturing expressions).
- ▷ To implement lazy evaluation, R uses a promise object that contains the expression needed to compute the result, and the environment in which to perform the computation.
- ▷ Creating these objects has some overhead, so every additional argument to an R function slows it down a little.
- ▷ The following microbenchmark compares the run time of a very simple function. Each version of the function has one extra argument. This suggests that each additional argument slows the function down by 20 ns.

```
f0 <- function() NULL
f1 <- function(a = 1) NULL
f2 <- function(a = 1, b = 1) NULL
f3 <- function(a = 1, b = 2, c = 3) NULL
f4 <- function(a = 1, b = 2, c = 4, d = 4) NULL
f5 <- function(a = 1, b = 2, c = 4, d = 4, e = 5) NULL
microbenchmark(f0(), f1(), f2(), f3(), f4(), f5(), times = 1000)

## Unit: nanoseconds
##  expr min  lq median    uq   max neval
##  f0() 251 267  272.0 317.0  1404  1000
##  f1() 286 322  332.0 382.0  1685  1000
##  f2() 321 366  387.0 459.0  1946  1000
##  f3() 361 432  472.0 582.5  1910  1000
##  f4() 397 469  527.0 637.0 14543  1000
##  f5() 452 543  627.5 738.0  2302  1000
```

- ▷ In most other programming languages there is little overhead for adding extra arguments. Many compiled languages will even warn you if arguments are never used (like in the above example), and automatically remove them from the function.

Implementation performance

◇ The design of R-language limits its maximum theoretical performance. But GNU-R is currently nowhere close to that maximum, and there are many things that can (and will) be done to speed it up.

◇ **Extracting a single value from a data frame**

- ▷ The following microbenchmark shows seven ways to access a single value (the number in the bottom-right corner) from the built-in mtcars dataset.
- ▷ The variation in performance is startling:
- ▷ There's no reason for there to be such a huge difference in performance, but no one has spent the time to make the slowest methods faster.

```
microbenchmark(mtcars[32, 11], mtcars[[c(11, 32)]], mtcars[[11]][32], mtcars$carb[32])

## Unit: microseconds
##                  expr    min     lq median     uq     max neval
##       mtcars[32, 11] 35.765 37.00 37.371 37.560   72.13   100
##  mtcars[[c(11, 32)]] 15.289 16.01 16.239 17.274   52.55   100
##     mtcars[[11]][32] 14.392 15.26 15.585 16.569   18.52   100
##      mtcars$carb[32]  4.357  5.62  5.943  6.069 3341.25   100
```

◇ ifelse(), pmin(), and pmax().

- ▷ the following three implementations of a function to squish() a vector to ensure that the smallest value is at least a and the largest value is at most b.
- ▷ squish_ife() uses ifelse(). ifelse() is known to be slow because it is relatively general, and must evaluate all arguments fully.
- ▷ squish_p(), uses pmin() and pmax() which should be much faster because they're so specialised. But they're actually rather slow because they can take any number of arguments and have to do some relatively complicated checks to determine which method to use.
- ▷ The final implementation uses basic subassignment.

```
squish_ife <- function(x, a, b) {
    ifelse(x <= a, a, ifelse(x >= b, b, x))
}
squish_p <- function(x, a, b) {
    pmax(pmin(x, b), a)
}
squish_in_place <- function(x, a, b) {
    x[x <= a] <- a
    x[x >= b] <- b
    x
}

x <- runif(100, -1.5, 1.5)
microbenchmark(squish_ife(x, -1, 1), squish_p(x, -1, 1), squish_in_place(x,
    -1, 1))

## Unit: microseconds
##                      expr    min     lq median     uq    max neval
##      squish_ife(x, -1, 1) 123.60 158.91 159.95 162.23 199.38   100
##        squish_p(x, -1, 1)  39.04  43.26  46.02  47.51 111.78   100
##  squish_in_place(x, -1, 1)  17.61  19.72  23.33  25.17  65.26   100
```

- ▷ There's quite a variation in speed: using pmin() and pmax() is about 3x faster than using ifelse(), and using subsetting directly is about twice as fast again.

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector squish_cpp(NumericVector x, double a, double b) {
int n = x.length();
NumericVector out(n);
for (int i = 0; i < n; ++i) {
double xi = x[i];
if (xi < a) {
out[i] = a;
} else if (xi > b) {
out[i] = b;
} else {
out[i] = xi;
}
}

return out;
}

microbenchmark(
squish_in_place(x, -1, 1),
squish_cpp(x, -1, 1)
)
```

Alternative R implementations

- The four most mature open-source projects are: (These are roughly ordered in from most practical to most ambitious.)
  - **pqR** (pretty quick R), by Radford Neal. It's built on top of the existing R code base (2.15.0), and fixes many obvious performance issues. It provides better memory management, and some support for automatic multithreading.http://www.pqr-project.org
  - **Renjin** by BeDataDriven. Renjin uses the java virtual machine, and has an extensive test suite. http://www.renjin.org/
  - **fastr,** by a team from Purdue. fastr is similar to Renjin, but it makes more ambitious optimisations and is somewhat less mature. https://github.com/allr/fastr
  - **Riposte**, by Justin Talbot and Zachary DeVito. Riposte is experimental and ambitious, and for the parts of R it implements is extremely fast. Riposte is described in more detail in Riposte: A Trace-Driven Compiler and Parallel VM for Vector Code in R. https://github.com/jtalbot/riposte
- However, even if these implementations never make a dent in the use of GNU-R, they have other benefits:
  - Simpler implementations make it easy to validate new approaches before porting to GNU-R.
  - Gain understanding about which aspects of language could be changed with minimal impact to existing code and maximal impact on performance.
  - Alternative implementations put pressure on the R-core to incorporate performance improvements.
- One of the most important approaches that pqR, renjin, fastR and riposte are exploring is the idea of deferred evaluation.
- As Justin Talbot, the author of riposte, points out "for long vectors, R's execution is completely memory bound. It spends almost all of its time reading and writing vector intermediates to memory". If you can eliminate intermediate vectors, you can not only decrease memory usage, you can also considerably improve performance.

◇ The following example shows a very simple example of where deferred evaluation might help. We have three vectors, x, y, z each containing 1 million elements, and we want to find the sum of x + y where z is TRUE.

```
x <- runif(1e+06)
y <- runif(1e+06)
z <- sample(c(T, F), 1e+06, rep = TRUE)
# sum((x + y)[z])
```

◇ In R, this creates two big temporary vectors: x + y, 1 million elements long, and (x + y)[z], about 500,000 elements long. This means you need to have extra memory free for the intermediate calculation, and you have to shuttle the data back and forth between the CPU and memory. This slows computation down because the CPU can't work at maximum efficiency if it's always waiting for more data to come in.

◇ If we rewrote the function using a loop in a language like C++, we could recognise that we only need one intermediate value: the sum of all the values we've seen:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]] double cond_sum_cpp(NumericVector x, NumericVector y, LogicalVector z) {
double sum = 0;
int n = x.length();
  for(int i = 0; i < n; i++) {
if (!z[i]) continue;
sum += x[i] + y[i];
}
  return sum;
}

cond_sum_r <- function(x, y, z) {
sum((x + y)[z])
}
microbenchmark(
cond_sum_cpp(x, y, z),
cond_sum_r(x, y, z),
unit = "ms" )
#> Unit: milliseconds
#>                  expr  min    lq median    uq  max neval
#>  cond_sum_cpp(x, y, z)  7.5  7.76   7.97  8.53 10.4   100
#>    cond_sum_r(x, y, z) 30.7 31.80  32.70 34.40 89.0   100
```

◇ The goal of deferred evaluation is to perform this transformation automatically, so you can write concise R code and have it automatically translated into efficient machine code.

5.2 Profiling[83]

◇ Your code should be correct, maintainable and fast. Notice that speed comes last - if your function is incorrect or unmaintainable (i.e. will eventually become incorrect) it doesn't matter if it's fast.

[83] "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil" — Donald Knuth.

- ◇ This often means vectorising code, or avoiding some of the most obvious traps discussed in the [R inferno]http://www.burns-stat.com/documents/books/the-r-inferno/.
- ◇ Making fast code is a four part process:

    1. **Profiling** helps you discover which parts of your code are taking up the most time.
    2. **Microbenchmarking** lets you experiment with small parts of your code to find faster approaches.
    3. **Timing** helps you check that the micro-optimisations have a macro effect, and helps experiment with larger changes (like totally rethinking your approach)
    4. **A performance testing tool** makes sure your code stays fast in the future (e.g. Vbench, http://wesmckinney.com/blog/?p=373)

- ◇ Sometimes there's no way to improve performance within R, and you'll need to use C++, the topic of Rcpp.
- ◇ Having a good test suite is important when tuning the performance of your code: you don't want to make your code fast at the expense of making it incorrect.
    - ▷ Good exploration from Winston: http://rpubs.com/wch/3797
    - ▷ Find out what is slow. Then make it fast.
    - ▷ Mature optimisation (PDF) http://carlos.bueno.org/optimization/mature-optimization.pdf
- ◇ A recurring theme throughout this part of the book is the importance of differentiating between absolute and relative speed, and fast vs fast enough.
    - ▷ First, whenever you compare the speed of two approaches to a problem, be very wary of just looking at a relative differences.
    - ▷ You also need to think about the costs of modifying your code. For example, if it takes you an hour to implement a change that makes you code 10x faster, saving 9 s each run, then you'll have to run at least 400 times before you'll see a net benefit.
    - ▷ Be careful that you don't spend hours to save seconds.

Performance profiling

Timing

Performance testing

Caching

Byte code compilation

Other people's code

Rewrite in a lower-level language

Brainstorming

5.3 Memory

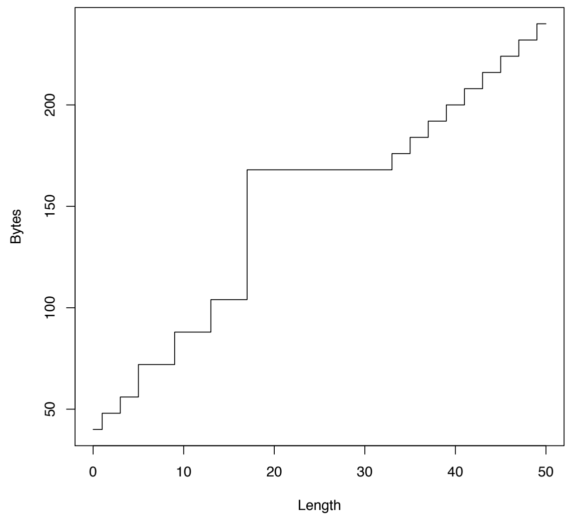
- ◇ Understanding how memory works in R can not only help you analyse larger datasets with the same amount of memory, but is also important for writing fast code, as accidental copies are a major cause of slow code.

- Places to learn memory management in R
  - reading of the documentation (particularly ?Memory and ?gc)
  - the memory profiling section of R-exts (http://cran.r-project.org/doc/manuals/R-exts.html#Profiling-R-code-for-memory-use)
  - the SEXPs section of R-ints. (http://cran.r-project.org/doc/manuals/R-ints.html#SEXPs)

object.size()

- One of the most useful tools for understanding memory usage in R is object.size(), which tells you how much memory an object occupies.
- You might have expected that the size of an empty vector would be 0 and that the memory usage would grow proportionately with length. Neither of those things are true!

```
sizes <- sapply(0:50, function(n) object.size(seq_len(n)))
plot(0:50, sizes, xlab = "Length", ylab = "Bytes", type = "s")
```



  - This isn't just an artefact of integer vectors: every vector of length 0 occupies 40 bytes of memory:

```
object.size(numeric())

## 40 bytes

object.size(logical())

## 40 bytes

object.size(raw())

## 40 bytes

object.size(list())

## 40 bytes
```

◇ What are those 40 bytes of memory used for? Every object in R has four components:

- ▷ **object metadata**, the *sxpinfo* (4 bytes). This metadata includes the base type, and information used for debugging and memory management.
- ▷ **Two pointers**: one to the next object in memory, and one to the previous object (2 * 8 bytes). This doubly-linked list makes it easy for internal R code to loop iterate through every object in memory.
- ▷ **A pointer to the attributes** (8 bytes).

◇ All vector types (e.g. atomic vectors and lists), have three more components:

- ▷ **The length of the vector** (4 bytes). Using 4 bytes should mean that R can only support vectors up to 2 ^ (4 * 8 - 1) (2 ^ 31, about two billion) elements long. But in R 3.0.0 and later you can have vectors up to 2 ^ 52 long: read R-internals to see how support for long vectors was added without changing the size of this field.
- ▷ **The "true" length of the vector** (4 bytes). This is basically never used, except when the object is the hash table for an environment, where the truelength represents the allocated space and the length represents the space currenty used.
- ▷ **The data** (?? bytes). An empty vector has 0 bytes of data, but it's obviously very important otherwise!

◇ If you're counting closely you'll note that only this adds up to 36 bytes. The other 4 bytes are needed as padding after the sxpinfo, so that the pointers start on 8 byte (=64-bit) boundaries.

◇ Most process architectures require this alignment for pointers, and even if not required, accessing non-aligned pointers tends to be rather slow. (If you're interested, you can read more about C structure package[84].)

◇ That explains the intercept on the graph.

◇ But why does the memory size grow in irregular jumps? To understand that, you need to know a little bit about how R requests memory from the operating system.

◇ Requesting memory, using the `malloc()` function, is a relatively expensive operation, and it would make R slow if it had to request memory every time you created a little vector.

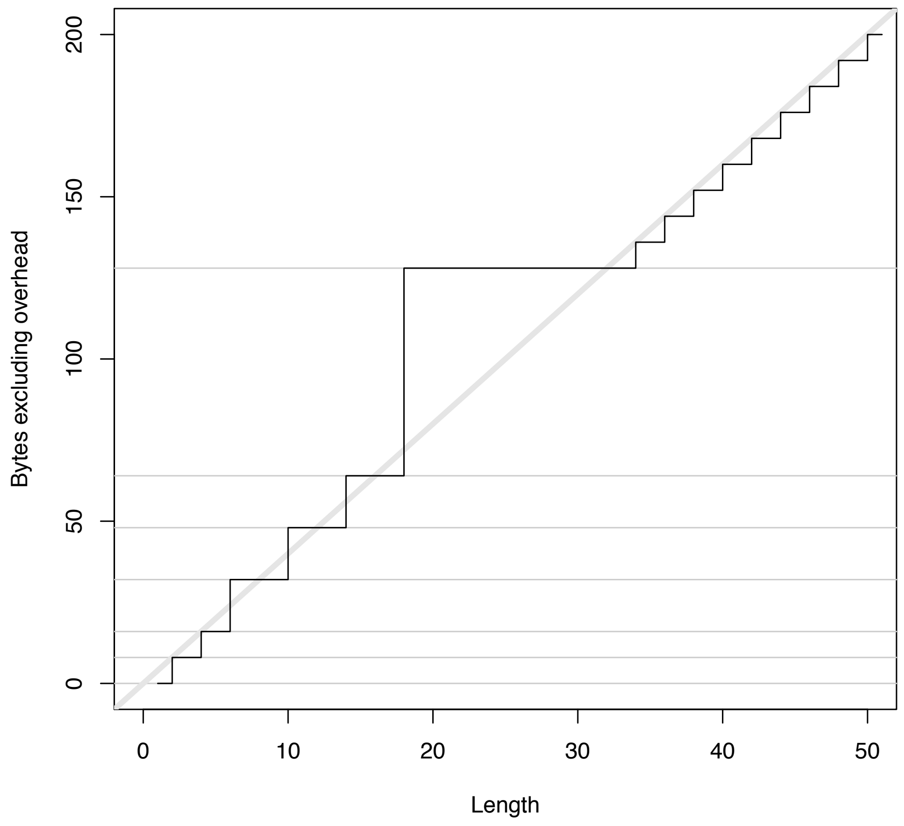
◇ Instead, it asks for a big block of memory and then manages it itself: this is called the **small vector pool**.

◇ R uses this pool for vectors less than 128 bytes long, and for efficiency and simplicity, it only allocates vectors that are 8, 16, 32, 48, 64 or 128 bytes long.

◇ If we adjust our previous plot by removing the 40 bytes of overhead we can see that those values correspond to the jumps.

```r
plot(0:50, sizes - 40, xlab = "Length", ylab = "Bytes excluding overhead", type = "n")
abline(h = 0, col = "grey80")
abline(h = c(8, 16, 32, 48, 64, 128), col = "grey80")
abline(a = 0, b = 4, col = "grey90", lwd = 4)
lines(sizes - 40, type = "s")
```

[84] http://www.catb.org/esr/structure-packing/



◇ It only remains to explain the steps after 128 bytes. While it makes sense for R to manage memory for small vectors, it doesn't make sense to manage it for large vectors: allocating big chunks of memory is something that operating systems are very good at.

◇ R always asks for memory in multiples of 8 bytes: this ensures good alignment for the data, in the same way we needed good alignment for the pointers.

◇ There are a few other subtleties to object.size(): it only promises to give an estimate of the memory usage, not the actual usage.

◇ This is because for more complex objects it's not immediately obvious what memory usage means.

◇ Take environments for example. Using object.size() on an environment tells you the size of the environment, not the size of its contents.

◇ It would be easy to create a function that did this:

```
env_size <- function(x) {
    if (!is.environment(x))
       return(object.size(x))

    objs <- ls(x, all = TRUE)
    sizes <- vapply(objs, function(o) env_size(get(o, x)), double(1))
    structure(sum(sizes) + object.size(x), class = "object_size")
}
object.size(environment())

## 56 bytes

env_size(environment())

## 20230792 bytes
```

◇ This function isn't quite correct because it's very difficult to cover every special case. For example, you might have an object with an attribute that's an environment that contains a formula which has an environment containing a large object...

◇ But even if you could cover all these special cases there's another problem. Environment objects are reference based so you can point to the same object from multiple locations.

◇ For example, In the following example, what should the size of b be?

```
a <- new.env()
a$x <- 1:1e+06
b <- new.env()
b$a <- a
env_size(a)

## 4000096 bytes

env_size(b)

## 4000152 bytes
```

You could argue that the size of b is actually only 56 bytes, because if you remove b, that's how much memory will be freed. But if you deleted a first, and then deleted b it would free 4000152 bytes. So is the size of b 56 or 4000152 bytes? The answer depends on the context.

◇ Another challenge for object.size() is strings:

```
object.size("banana")

## 96 bytes

object.size(rep("banana", 100))

## 888 bytes
```

▷ the size of a vector containing "banana" is 96 bytes, but the size of a vector containing 100 "banana"s is 888 bytes. Why the difference? The key is 888 = 96 + 99 * 8.

- ▷ R has a global string pool, which means that every unique string is only stored once in memory. Every other instance of that string is just a pointer, and only needs 8 bytes of storage.
- ▷ object.size() does tries to take this into account for individual vectors, but like with environments it's not obvious exactly how the accounting should work.

Total memory use

- ◇ object.size() tells you the size of a single object; gc() (among other things) tells you the total size of all objects in memory:

```
gc()

##           used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  197970 10.6     407500 21.8   407500 21.8
## Vcells 3456110 26.4    4989434 38.1  4442871 33.9
```

- ◇ R breaks down memory usage into **Vcells** (memory used by vectors) and **Ncells** (memory used by everything else).
- ◇ But this distinction isn't usually important, and neither are the gc trigger and max used columns. What you're usually most interested in is the total memory used.
- ◇ The function below wraps around gc() to return just the amount of memory (in megabytes) that R is currently using.

```
rm(list = ls())
mem <- function() {
    bit <- 8L * .Machine$sizeof.pointer
    if (!(bit == 32L || bit == 64L)) {
        stop("Unknown architecture", call. = FALSE)
    }

    node_size <- if (bit == 32L)
        28L else 56L
    usage <- gc()
    sum(usage[, 1] * c(node_size, 8))/(1024^2)
}
mem()

## [1] 13.88
```

- ◇ Don't expect this number to agree with the amount of memory that your operating system says that R is using:
  - ▷ Some overhead associated with the R interpreter is not captured by these numbers.
  - ▷ Both R and the operating system are lazy: they won't try and reclaim memory until it's actually needed. So R might be holding on to memory because the OS hasn't asked for it back yet.
  - ▷ R counts the memory occupied by objects; there may be gaps from objects that have been deleted. This problem is known as memory fragmentation.

◇ We can build a function on top of mem() that tells us how memory changes during the execution of a block of code. Positive numbers represent an increase in the memory used by R, and negative numbers a decrease.

```
mem_change <- function(code) {
    start <- mem()
    expr <- substitute(code)
    eval(expr, parent.frame())
    rm(code, expr)

    round(mem() - start, 3)
}
# Need about 4 mb to store 1 million integers
mem_change(x <- 1:1e+06)

## [1] 3.814

# We get that memory back when we delete it
mem_change(rm(x))

## [1] -3.815
```

Garbage collection

◇ In some languages you have to explicitly delete unnused objects so that their memory can be returned. R uses an alternative approach, called **garbage collection** (GC for short), which automatically released memory when an object is no longer used.

◇ It does this based on environments and the regular scoping rules: when an environment goes out of scope (for example, when a function finishes executing), all of the contents of that environment are deleted and their memory is freed.

```
f <- function() {
    1:1e+06
}
mem_change(f())

## [1] 0
```

◇ This is a little bit of a simplification because in order to find out how much memory is available, our mem() function calls gc().

◇ As well as returning the amount of memory currently used, gc() also triggers garbage collection.

◇ Garbage collection normally happens lazily: R calls gc() when it needs more space. In reality, that R might hold onto the memory after the function has terminated, but it will release it as soon as it's needed.

◇ Despite what you might have read elsewhere, there's never any point in calling gc() yourself, apart to see how much memory is in use.

◇ R will automatically run garbage collection whenever it needs more space; if you want to see when that is, call gcinfo(TRUE).

◇ The only reason you might want to call gc() is that it also requests that R should return memory to the operating system.

◇ Generally, GC takes care of releasing previously used memory. However, you do need to be aware of situations that can cause memory leaks: when you think you've removed all references to an object, but some are still hanging around so the object never gets freed.

◇ In R, the two main causes of memory leaks are formulas and closures. They both capture the enclosing environment, so objects in that environment will not be reclaimed automatically.

```
f1 <- function() {
    x <- 1:1e+06
    10
}
mem_change(x <- f1())

## [1] 0

x

## [1] 10

rm(x)

f2 <- function() {
    x <- 1:1e+06
    a ~ b
}
mem_change(y <- f2())

## [1] 3.815

object.size(y)

## 712 bytes

rm(y)

f3 <- function() {
    x <- 1:1e+06
    function() 10
}
mem_change(z <- f3())

## [1] 3.814

object.size(z)

## 936 bytes

rm(z)
```

Memory profiling with lineprof

◇ As well as using mem_change() to explicitly capture the change in memory caused by running a block of code, we can use memory profiling to automatically capture memory usage every few milliseconds.

◇ This functionality is provided by the utils::Rprof(), but it doesn't provide a very useful display of the results. Instead, we'll use the lineprof[85] package; it's powered by Rprof(), but displays the results in a more informative manner.

Modification in place

◇ What happens to x in the following code?

```
x <- 1:10
x[5] <- 10
x

##  [1]  1  2  3  4 10  6  7  8  9 10
```

There's two possibilities:

1. R modifies the existing x in place.
2. R makes a copy of x in a new location, modifies that new vector, and then changes the name x to point to the new location.

◇ It turns out that R can do either depending on the circumstances.

◇ In the example above, it will modify in place, but if another variable also points to x, then it will copy it to a new location: To explore what's going on in more detail we need some new tools found in the pryr package.

◇ Given the name of a variable, address() tells us its location in memory, and refs() tells us how many names point to that same location.[86]

```
library(pryr)

## Loading required package:  Rcpp
##
## Attaching package:  'pryr'
##
## The following object is masked _by_ '.GlobalEnv':
##
##    f

x <- 1:10
c(address(x), refs(x))

## [1] "0x1012fd038" "2"
```

[85]https://github.com/hadley/lineprof
[86]Note that if you're using Rstudio this refs() will always return two: the environment browser makes a reference to every object you create on the command line, but not inside a function.

```
y <- x
c(address(y), refs(y))

## [1] "0x1012fd038" "2"
```

◇ Note that refs is only an estimate and it can only distinguish between 1 and more than 1 references. This means that refs() returns 2 in both of the following cases:

```
x <- 1:5
y <- x
rm(y)
# Should really be one, because we've deleted y
refs(x)

## [1] 2

x <- 1:5
y <- x
z <- x
# Should really be three
refs(x)

## [1] 2
```

◇ When refs(x) is one, modification will occur in place; when refs(x) is two, it will make a copy (so that the other pointers to the object contined unchanged).

```
x <- 1:10
y = x
c(address(x), address(y))

## [1] "0x103dbb670" "0x103dbb670"

x[5] <- 6L
c(address(x), address(y))

## [1] "0x100c296c8" "0x103dbb670"
```

◇ Another useful function is tracemem(), which will print a message every time the traced object is copied:

```
x <- 1:10
# Prints the current memory location of the object
tracemem(x)

## [1] "<0x10371b978>"

y <- x
# Prints where it has moved from and to
x[5] <- 6L
```

```
## tracemem[0x10371b978 -> 0x10385ad88]: eval eval withVisible withCallingHandlers doTryCatch tryCa
```

◇ Non-primitive functions that touch the object always increment the ref count. Primitive functions are usually written in such a way that they don't increment the ref count.[87]

```
rm(list = ls())
x <- 1:10
refs(x)

## [1] 2

mean(x)

## [1] 5.5

refs(x)

## [1] 2

# Touching the object forces an increment
f <- function(x) x
x <- 1:10
f(x)

##  [1]  1  2  3  4  5  6  7  8  9 10

refs(x)

## [1] 2

# Sum is primitive, so doesn't increment
x <- 1:10
sum(x)

## [1] 55

refs(x)

## [1] 2

# f() and g() never evaluate x so refs doesn't increment
f <- function(x) 10
x <- 1:10
f(x)

## [1] 10
```

[87]The reasons are a little complicated, but see the R-devel thread confused about NAMEDhttp://r.789695.n4.nabble.com/Confused-about-NAMED-td4103326.html

```r
refs(x)

## [1] 2

g <- function(x) substitute(x)
x <- 1:10
g(x)

## x

refs(x)

## [1] 2
```

- Generally, any primitive replacement function will modify in place, provided that the object is not referred to elsewhere. This includes [[<-, [<-, @<-, $<-, attr<-, attributes<-, class<-, dim<-, dimnames<-, names<-, and levels<-.
- To be precise, all non-primitive functions increment refs, but a primitive function may be written in such a way that it doesn't increment refs.
- The rules are sufficiently complicated that there's not a lot of point in trying to memorise them; instead approach the problem practically; use refs() and tracemem() to figure out when objects are being copied.
- If you find yourself resorting to exotic tricks to avoid copies, it may be time to rewrite your function in C++, as described in Rcpp.
- **Loops**
  - For loops in R have a reputation for being slow, but often this slowness is because instead of modifying in place, you're modifying a copy.
  - Take the following code that subtracts the median from each column of a large data.frame:

```r
rm(list = ls())
x <- data.frame(matrix(runif(100 * 10000), ncol = 100))
medians <- vapply(x, median, numeric(1))
system.time({
    for (i in seq_along(medians)) {
        x[, i] <- x[, i] - medians[i]
    }
})

##    user  system elapsed
##   8.076   0.540   8.696
```

  - It's rather slow - we only have 100 columns and 10,000 rows, but it's taking almost eight seconds. We can use address() and refs() to see what's going on for a small sample of the loop:

```r
for (i in 1:5) {
    x[, i] <- x[, i] - medians[i]
    print(c(address(x), refs(x)))
}
```

```
## [1] "0x100628a80" "2"
## [1] "0x100760880" "2"
## [1] "0x10075f260" "2"
## [1] "0x100760880" "2"
## [1] "0x10075f260" "2"
```

- ▷ In each iteration x is moved to a new location (copying the complete data frame) and refs(x) is always 2. This is because [<-.data.frame is not a primitive function, so it always increments the refs.
- ▷ We can make the function substantially more efficient by using either a list or matrix instead of a data frame. Modifying lists and matrices use primitive functions, so the refs are not incremented and all modifications are in place.

```
y <- as.list(x)
system.time({
    for (i in seq_along(medians)) {
        y[[i]] <- y[[i]] - medians[i]
    }
})

##    user  system elapsed
##   0.012   0.001   0.012

z <- as.matrix(x)
system.time({
    for (i in seq_along(medians)) {
        z[, i] <- z[, i] - medians[i]
    }
})

##    user  system elapsed
##   0.118   0.003   0.122
```

5.4 Rcpp

Getting started

Getting started with C++

Rcpp classes and methods

Rcpp sugar

Missing values

The STL

Case studies

Using Rcpp in a Package

Adding Rcpp to an existing package (Rcpp<=0.10.6)

Adding Rcpp to an existing package (Rcpp>=0.10.7)

More details

Learning more

Acknowledgements

5.5 R's C interface

Introduction

Differences between R and C

Calling C functions from R

Basic data structures

Coercion and object creation

Modifying objects

Pairlists and symbols

Missing and symbols

Missing and non-finite values

Checking types in C

Finding the C source code for a function .External

Using C code in a package

Part II
Packages

6 Getting started

6.1 Philosophy

Getting started

Introduction to devtools

6.2 Package basics

6.3 Package development cycle

6.4 Quick ref

7 Documentation

7.1 Documenting packages

7.2 Documenting functions

7.3 Namespaces

8 Best practices

8.1 Good code style

8.2 Testing

8.3 Git and github

8.4 Release