Exercises from Data-structures.rmd

Exercises

1. What are the six types of atomic vector? How does a list differ from an atomic vector?

2. What makes `is.vector()` and `is.numeric()` fundamentally different to `is.list()` and `is.character()`?

3. Test your knowledge of vector coercion rules by predicting the output of the following uses of `c()`:

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

4. Why do you need to use `unlist()` to convert a list to an atomic vector? Why doesn't `as.vector()` work?

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

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

Exercises

1. An early draft used this code to illustrate `structure()`:

```
structure(1:5, comment = "my attribute")
```

```
## [1] 1 2 3 4 5
```

   But when you print that object you don't see the comment attribute. Why? Is the attribute missing, or is there something else special about it? (Hint: try using help.) \index{attributes!comment}

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

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

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

```
f2 <- rev(factor(letters))

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

Exercises

1. What does `dim()` return when applied to a vector?
2. If `is.matrix(x)` is `TRUE`, what will `is.array(x)` return?
3. How would you describe the following three objects? What makes them different to `1:5`?

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

Exercises

1. What attributes does a data frame possess?
2. What does `as.matrix()` do when applied to a data frame with columns of different types?
3. Can you have a data frame with 0 rows? What about 0 columns?

Exercises from Subsetting.rmd

Exercises

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

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

2. Why does `x <- 1:5; x[NA]` yield five missing values? (Hint: why is it different from `x[NA_real_]`?)
3. What does `upper.tri()` return? How does subsetting a matrix with it work? Do we need any additional subsetting rules to describe its behaviour?

```
x <- outer(1:5, 1:5, FUN = "*")
x[upper.tri(x)]
```

4. Why does `mtcars[1:20]` return an error? How does it differ from the similar

`mtcars[1:20, ]`?

5. Implement your own function that extracts the diagonal entries from a matrix (it should behave like `diag(x)` where `x` is a matrix).

6. What does `df[is.na(df)] <- 0` do? How does it work?

Exercises

1. Given a linear model, e.g., `mod <- lm(mpg ~ wt, data = mtcars)`, extract the residual degrees of freedom. Extract the R squared from the model summary (`summary(mod)`)

Exercises

1. How would you randomly 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?

2. 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)?

3. How could you put the columns in a data frame in alphabetical order?

Exercises from Functions.rmd

Exercises

1. What function allows you to tell if an object is a function? What function allows you to tell if a function is a primitive function?

2. This code makes a list of all functions in the base package.

```
objs <- mget(ls("package:base"), inherits = TRUE)
funs <- Filter(is.function, objs)
```

   Use it to answer the following questions:

   a. Which base function has the most arguments?

   a. How many base functions have no arguments? What's special about those functions?

   a. How could you adapt the code to find all primitive functions?

3. What are the three important components of a function?

4. When does printing a function not show what environment it was created in?

Exercises

1. What does the following code return? Why? What does each of the three `c`'s mean?

```
c <- 10
c(c = c)
```

2. What are the four principles that govern how R looks for values?

3. What does the following function return? Make a prediction before running the code yourself.

```
f <- function(x) {
  f <- function(x) {
    f <- function(x) {
      x ^ 2
    }
    f(x) + 1
  }
  f(x) * 2
}
f(10)
```

Exercises

1. Clarify the following list of odd function calls:

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

2. What does this function return? Why? Which principle does it illustrate?

```
f1 <- function(x = {y <- 1; 2}, y = 0) {
  x + y
}
f1()
```

3. What does this function return? Why? Which principle does it illustrate?

```
f2 <- function(x = z) {
  z <- 100
  x
}
f2()
```

Exercises

1. Create a list of all the replacement functions found in the base package. Which ones are primitive functions?

2. What are valid names for user-created infix functions?

3. Create an infix `xor()` operator.

4. Create infix versions of the set functions `intersect()`, `union()`, and `setdiff()`.

5. Create a replacement function that modifies a random location in a vector.

Exercises

1. How does the `chdir` parameter of `source()` compare to `in_dir()`? Why might you prefer one approach to the other?

2. What function undoes the action of `library()`? How do you save and restore the values of `options()` and `par()`?

3. Write a function that opens a graphics device, runs the supplied code, and closes the graphics device (always, regardless of whether or not the plotting code worked).

4. We can use `on.exit()` to implement a simple version of `capture.output()`.

```
capture.output2 <- function(code) {
  temp <- tempfile()
  on.exit(file.remove(temp), add = TRUE)

  sink(temp)
  on.exit(sink(), add = TRUE)

  force(code)
  readLines(temp)
}
capture.output2(cat("a", "b", "c", sep = "\n"))
```

```
## Warning in file.remove(temp): cannot remove file 'C:\Users\richad\AppD
## \Local\Temp\RtmpwjevHF\file1a1c3bcf87', reason 'Permission denied'
```

```
## [1] "a" "b" "c"
```

Compare `capture.output()` to `capture.output2()`. How do the functions differ? What features have I removed to make the key ideas easier to see? How have I rewritten the key ideas to be easier to understand?

Exercises from OO-essentials.rmd

Exercises

1. Read the source code for `t()` and `t.test()` and confirm that `t.test()` is an S3 generic and not an S3 method. What happens if you create an object with class `test` and call `t()` with it?

2. What classes have a method for the `Math` group generic in base R? Read the source code. How do the methods work?

3. R has two classes for representing date time data, `POSIXct` and `POSIXlt`, which both inherit from `POSIXt`. Which generics have different behaviours for the two classes? Which generics share the same behaviour?

4. Which base generic has the greatest number of defined methods?

5. `UseMethod()` calls methods in a special way. Predict what the following code will return, then run it and read the help for `UseMethod()` to figure out what's going on. Write down the rules in the simplest form possible.

```
y <- 1
g <- function(x) {
  y <- 2
  UseMethod("g")
}
g.numeric <- function(x) y
g(10)

h <- function(x) {
  x <- 10
  UseMethod("h")
}
h.character <- function(x) paste("char", x)
h.numeric <- function(x) paste("num", x)

h("a")
```

6. Internal generics don't dispatch on the implicit class of base types. Carefully read `?"internal generic"` to determine why the length of `f` and `g` is different in the example below. What function helps distinguish between the behaviour of `f` and `g`?

```
f <- function() 1
g <- function() 2
class(g) <- "function"

class(f)
class(g)

length.function <- function(x) "function"
length(f)
length(g)
```

Exercises

1. Which S4 generic has the most methods defined for it? Which S4 class has the most methods associated with it?

2. What happens if you define a new S4 class that doesn't "contain" an existing class? (Hint: read about virtual classes in `?Classes`.)

3. What happens if you pass an S4 object to an S3 generic? What happens if you pass an S3 object to an S4 generic? (Hint: read `?setOldClass` for the second case.)

Exercises

1. Use a field function to prevent the account balance from being directly manipulated. (Hint: create a "hidden" `.balance` field, and read the help for the fields argument in `setRefClass()`.)

2. I claimed that there aren't any RC classes in base R, but that was a bit of a simplification. Use `getClasses()` and find which classes `extend()` from `envRefClass`. What are the classes used for? (Hint: recall how to look up the documentation for a class.)

Exercises from Environments.rmd

Exercises

1. List three ways in which an environment differs from a list.

2. If you don't supply an explicit environment, where do `ls()` and `rm()` look? Where does `<-` make bindings?

3. Using `parent.env()` and a loop (or a recursive function), verify that the ancestors of `globalenv()` include `baseenv()` and `emptyenv()`. Use the same basic idea to implement your own version of `search()`.

Exercises

1. Modify `where()` to find all environments that contain a binding for `name`.

2. Write your own version of `get()` using a function written in the style of `where()`.

3. Write a function called `fget()` that finds only function objects. It should have two arguments, `name` and `env`, and should obey the regular scoping rules for functions: if there's an object with a matching name that's not a function, look in the parent. For an added challenge, also add an `inherits` argument which controls whether the function recurses up the parents or only looks in one environment.

4. Write your own version of `exists(inherits = FALSE)` (Hint: use `ls()`.) Write a recursive version that behaves like `exists(inherits = TRUE)`.

Exercises

1. List the four environments associated with a function. What does each one do? Why is the distinction between enclosing and binding environments particularly important?

2. Draw a diagram that shows the enclosing environments of this function:

```
f1 <- function(x1) {
  f2 <- function(x2) {
    f3 <- function(x3) {
      x1 + x2 + x3
    }
    f3(3)
  }
  f2(2)
}
f1(1)
```

3. Expand your previous diagram to show function bindings.

4. Expand it again to show the execution and calling environments.

5. Write an enhanced version of `str()` that provides more information about functions. Show where the function was found and what environment it was defined in.

Exercises

1. What does this function do? How does it differ from `<<-` and why might you prefer it?

```
rebind <- function(name, value, env = parent.frame()) {
  if (identical(env, emptyenv())) {
    stop("Can't find ", name, call. = FALSE)
  } else if (exists(name, envir = env, inherits = FALSE)) {
    assign(name, value, envir = env)
  } else {
    rebind(name, value, parent.env(env))
  }
}
rebind("a", 10)
```

```
## Error: Can't find a
```

```
a <- 5
rebind("a", 10)
a
```

```
## [1] 10
```

2. Create a version of `assign()` that will only bind new names, never re-bind old names. Some programming languages only do this, and are known as [single assignment languages][single assignment].

3. Write an assignment function that can do active, delayed, and locked bindings. What might you call it? What arguments should it take? Can you guess which sort of assignment it should do based on the input?

Exercises from Exceptions-Debugging.rmd

Exercises

- Compare the following two implementations of `message2error()`. What is the main advantage of `withCallingHandlers()` in this scenario? (Hint: look carefully at the traceback.)

```
message2error <- function(code) {
  withCallingHandlers(code, message = function(e) stop(e))
}
message2error <- function(code) {
  tryCatch(code, message = function(e) stop(e))
}
```

Exercises

- The goal of the `col_means()` function defined below is to compute the means of all numeric columns in a data frame.

```
col_means <- function(df) {
  numeric <- sapply(df, is.numeric)
  numeric_cols <- df[, numeric]

  data.frame(lapply(numeric_cols, mean))
}
```

  However, the function is not robust to unusual inputs. Look at the following results, decide which ones are incorrect, and modify `col_means()` to be more robust. (Hint: there are two function calls in `col_means()` that are particularly prone to problems.)

```
col_means(mtcars)
col_means(mtcars[, 0])
col_means(mtcars[0, ])
col_means(mtcars[, "mpg", drop = F])
col_means(1:10)
col_means(as.matrix(mtcars))
col_means(as.list(mtcars))

mtcars2 <- mtcars
mtcars2[-1] <- lapply(mtcars2[-1], as.character)
col_means(mtcars2)
```

- The following function “lags” a vector, returning a version of `x` that is `n` values behind the original. Improve the function so that it (1) returns a useful error message if `n` is not a vector, and (2) has reasonable behaviour when `n` is 0 or longer than `x`.

```
lag <- function(x, n = 1L) {
  xlen <- length(x)
  c(rep(NA, n), x[seq_len(xlen - n)])
}
```

Exercises from Functional-programming.rmd

Exercises

1. Given a function, like `"mean"`, `match.fun()` lets you find a function. Given a function, can you find its name? Why doesn't that make sense in R?

2. Use `lapply()` and an anonymous function to find the coefficient of variation (the standard deviation divided by the mean) for all columns in the `mtcars` dataset.

3. Use `integrate()` and an anonymous function to find the area under the curve for the

following functions. Use Wolfram Alpha to check your answers.

1. `y = x ^ 2 - x`, x in [0, 10]
2. `y = sin(x) + cos(x)`, x in [-π, π]
3. `y = exp(x) / x`, x in [10, 20]

4. A good rule of thumb is that an anonymous function should fit on one line and shouldn't need to use `{}`. Review your code. Where could you have used an anonymous function instead of a named function? Where should you have used a named function instead of an anonymous function?

Exercises

1. Why are functions created by other functions called closures?

2. What does the following statistical function do? What would be a better name for it? (The existing name is a bit of a hint.)

```
bc <- function(lambda) {
  if (lambda == 0) {
    function(x) log(x)
  } else {
    function(x) (x ^ lambda - 1) / lambda
  }
}
```

3. What does `approxfun()` do? What does it return?

4. What does `ecdf()` do? What does it return?

5. Create a function that creates functions that compute the ith central moment of a numeric vector. You can test it by running the following code:

```
m1 <- moment(1)
m2 <- moment(2)

x <- runif(100)
stopifnot(all.equal(m1(x), 0))
stopifnot(all.equal(m2(x), var(x) * 99 / 100))
```

6. Create a function `pick()` that takes an index, `i`, as an argument and returns a function with an argument `x` that subsets `x` with `i`.

```
lapply(mtcars, pick(5))
# should do the same as this
lapply(mtcars, function(x) x[[5]])
```

Exercises

1. Implement a summary function that works like `base::summary()`, but uses a list of functions. Modify the function so it returns a closure, making it possible to use it as a function factory.

2. Which of the following commands is equivalent to `with(x, f(z))`?

   (a) `x$f(x$z)`. (b) `f(x$z)`. © `x$f(z)`. (d) `f(z)`. (e) It depends.

Exercises

1. Instead of creating individual functions (e.g., `midpoint()`, `trapezoid()`, `simpson()`, etc.), we could store them in a list. If we did that, how would that change the code? Can you create the list of functions from a list of coefficients for the Newton-Cotes formulae?

2. The trade-off between integration rules is that more complex rules are slower to compute, but need fewer pieces. For `sin()` in the range [0, π], determine the number of pieces needed so that each rule will be equally accurate. Illustrate your results with a graph. How do they change for different functions? `sin(1 / x^2)` is particularly challenging.

Exercises from Functionals.rmd

Exercises

1. Why are the following two invocations of `lapply()` equivalent?

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(100)

lapply(trims, function(trim) mean(x, trim = trim))
lapply(trims, mean, x = x)
```

2. The function below scales a vector so it falls in the range [0, 1]. How would you apply it to every column of a data frame? How would you apply it to every numeric column in a data frame?

```
scale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

3. Use both for loops and `lapply()` to fit linear models to the `mtcars` using the formulas stored in this list:

```
formulas <- list(
  mpg ~ disp,
  mpg ~ I(1 / disp),
  mpg ~ disp + wt,
  mpg ~ I(1 / disp) + wt
)
```

4. Fit the model `mpg ~ disp` to each of the bootstrap replicates of `mtcars` in the list below by using a for loop and `lapply()`. Can you do it without an anonymous function?

```
bootstraps <- lapply(1:10, function(i) {
  rows <- sample(1:nrow(mtcars), rep = TRUE)
  mtcars[rows, ]
})
```

5. For each model in the previous two exercises, extract R^2 using the function below.

```
rsq <- function(mod) summary(mod)$r.squared
```

Exercises

1. Use `vapply()` to:

   a) Compute the standard deviation of every column in a numeric data frame.

   a) Compute the standard deviation of every numeric column in a mixed data frame. (Hint: you'll need to use `vapply()` twice.)

2. Why is using `sapply()` to get the `class()` of each element in a data frame dangerous?

3. The following code simulates the performance of a t-test for non-normal data. Use `sapply()` and an anonymous function to extract the p-value from every trial.

```
trials <- replicate(
  100,
  t.test(rpois(10, 10), rpois(7, 10)),
  simplify = FALSE
)
```

   Extra challenge: get rid of the anonymous function by using `[[` directly.

4. What does `replicate()` do? What sort of for loop does it eliminate? Why do its arguments differ from `lapply()` and friends?

5. Implement a version of `lapply()` that supplies `FUN` with both the name and the value of each component.

6. Implement a combination of `Map()` and `vapply()` to create an `lapply()` variant that iterates in parallel over all of its inputs and stores its outputs in a vector (or a matrix). What arguments should the function take?

7. Implement `mcsapply()`, a multicore version of `sapply()`. Can you implement `mcvapply()`, a parallel version of `vapply()`? Why or why not?

Exercises

1. How does `apply()` arrange the output? Read the documentation and perform some experiments.

2. There's no equivalent to `split()` + `vapply()`. Should there be? When would it be useful? Implement one yourself.

3. Implement a pure R version of `split()`. (Hint: use `unique()` and subsetting.) Can you do it without a for loop?

4. What other types of input and output are missing? Brainstorm before you look up some answers in the plyr paper.

Exercises

1. Why isn't `is.na()` a predicate function? What base R function is closest to being a predicate version of `is.na()`?

2. Use `Filter()` and `vapply()` to create a function that applies a summary statistic to every numeric column in a data frame.

3. What's the relationship between `which()` and `Position()`? What's the relationship between `where()` and `Filter()`?

4. Implement `Any()`, a function that takes a list and a predicate function, and returns `TRUE` if the predicate function returns `TRUE` for any of the inputs. Implement `All()` similarly.

5. Implement the `span()` function from Haskell: given a list `x` and a predicate function `f`, `span` returns the location of the longest sequential run of elements where the predicate is true. (Hint: you might find `rle()` helpful.)

Exercises

1. Implement `arg_max()`. It should take a function and a vector of inputs, and return the elements of the input where the function returns the highest value. For example, `arg_max(-10:5, function(x) x ^ 2)` should return -10. `arg_max(-5:5, function(x) x ^ 2)` should return `c(-5, 5)`. Also implement the matching `arg_min()` function.

2. Challenge: read about the fixed point algorithm. Complete the exercises using R.

Exercises

1. Implement `smaller` and `larger` functions that, given two inputs, return either the smaller or the larger value. Implement `na.rm = TRUE`: what should the identity be? (Hint: `smaller(x, smaller(NA, NA, na.rm = TRUE), na.rm = TRUE)` must be `x`, so `smaller(NA, NA, na.rm = TRUE)` must be bigger than any other value of x.) Use `smaller` and `larger` to implement equivalents of `min()`, `max()`, `pmin()`, `pmax()`, and new functions `row_min()` and `row_max()`.

2. Create a table that has *and*, *or*, *add*, *multiply*, *smaller*, and *larger* in the columns and *binary operator*, *reducing variant*, *vectorised variant*, and *array variants* in the rows.

   a) Fill in the cells with the names of base R functions that perform each of the roles.

   a) Compare the names and arguments of the existing R functions. How consistent are they? How could you improve them?

   a) Complete the matrix by implementing any missing functions.

3. How does `paste()` fit into this structure? What is the scalar binary function that underlies `paste()`? What are the `sep` and `collapse` arguments to `paste()` equivalent to? Are there any `paste` variants that don't have existing R implementations?

Exercises from Function-operators.rmd

Exercises

1. Write a FO that logs a time stamp and message to a file every time a function is run.

2. What does the following function do? What would be a good name for it?

```
f <- function(g) {
  force(g)
  result <- NULL
  function(...) {
    if (is.null(result)) {
      result <<- g(...)
    }
    result
  }
}
runif2 <- f(runif)
runif2(5)
```

```
## [1] 0.8193771 0.9522579 0.7767082 0.6220122 0.2986077
```

```
runif2(10)
```

```
## [1] 0.8193771 0.9522579 0.7767082 0.6220122 0.2986077
```

3. Modify `delay_by()` so that instead of delaying by a fixed amount of time, it ensures that a certain amount of time has elapsed since the function was last called. That is, if you called `g <- delay_by(1, f); g(); Sys.sleep(2); g()` there shouldn't be an extra delay.

4. Write `wait_until()` which delays execution until a specific time.

5. There are three places we could have added a memoise call: why did we choose the one we did?

```
download <- memoise(dot_every(10, delay_by(1, download_file)))
download <- dot_every(10, memoise(delay_by(1, download_file)))
download <- dot_every(10, delay_by(1, memoise(download_file)))
```

6. Why is the `remember()` function inefficient? How could you implement it in more efficient way?

7. Why does the following code, from stackoverflow, not do what you expect?

```
# return a linear function with slope a and intercept b.
f <- function(a, b) function(x) a * x + b

# create a list of functions with different parameters.
fs <- Map(f, a = c(0, 1), b = c(0, 1))

fs[[1]](3)
```

```
## [1] 4
```

```
# should return 0 * 3 + 0 = 0
```

   How can you modify `f` so that it works correctly?

Exercises

1. Create a `negative()` FO that flips the sign of the output of the function to which it is applied.

2. The `evaluate` package makes it easy to capture all the outputs (results, text, messages, warnings, errors, and plots) from an expression. Create a function like `capture_it()` that also captures the warnings and errors generated by a function.

3. Create a FO that tracks files created or deleted in the working directory (Hint: use `dir()` and `setdiff()`.) What other global effects of functions might you want to track?

Exercises

1. Our previous `download()` function only downloads a single file. How can you use `partial()` and `lapply()` to create a function that downloads multiple files at once? What are the pros and cons of using `partial()` vs. writing a function by hand?

2. Read the source code for `plyr::colwise()`. How does the code work? What are `colwise()`'s three main tasks? How could you make `colwise()` simpler by implementing each task as a function operator? (Hint: think about `partial()`.)

3. Write FOs that convert a function to return a matrix instead of a data frame, or a data frame instead of a matrix. If you understand S3, call them `as.data.frame.function()` and `as.matrix.function()`.

4. You've seen five functions that modify a function to change its output from one form to another. What are they? Draw a table of the various combinations of types of outputs: what should go in the rows and what should go in the columns? What function operators might you want to write to fill in the missing cells? Come up with example use cases.

5. Look at all the examples of using an anonymous function to partially apply a function in this and the previous chapter. Replace the anonymous function with `partial()`. What do you think of the result? Is it easier or harder to read?

Exercises

1. Implement your own version of `compose()` using `Reduce` and `%o%`. For bonus points, do it without calling `function`.

2. Extend `and()` and `or()` to deal with any number of input functions. Can you do it with `Reduce()`? Can you keep them lazy (e.g., for `and()`, the function returns once it sees the first `FALSE`)?

3. Implement the `xor()` binary operator. Implement it using the existing `xor()` function. Implement it as a combination of `and()` and `or()`. What are the advantages and disadvantages of each approach? Also think about what you'll call the resulting function to avoid a clash with the existing `xor()` function, and how you might change the names of `and()`, `not()`, and `or()` to keep them consistent.

4. Above, we implemented boolean algebra for functions that return a logical function. Implement elementary algebra (`plus()`, `minus()`, `multiply()`, `divide()`, `exponentiate()`, `log()`) for functions that return numeric vectors.

Exercises from Computing-on-the-language.rmd

Exercises

1. One important feature of `deparse()` to be aware of when programming is that it can return multiple strings if the input is too long. For example, the following call produces a vector of length two:

   ```
   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)
   ```

   Why does this happen? Carefully read the documentation. Can you write a wrapper around `deparse()` so that it always returns a single string?

2. Why does `as.Date.default()` use `substitute()` and `deparse()`? Why does `pairwise.t.test()` use them? Read the source code.

3. `pairwise.t.test()` assumes that `deparse()` always returns a length one character vector. Can you construct an input that violates this expectation? What happens?

4. `f()`, defined above, just calls `substitute()`. Why can't we use it to define `g()`? In other words, what will the following code return? First make a prediction. Then run the code and think about the results.

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

Exercises

1. Predict the results of the following lines of code:

   ```
   eval(quote(eval(quote(eval(quote(2 + 2))))))
   eval(eval(quote(eval(quote(eval(quote(2 + 2)))))))
   quote(eval(quote(eval(quote(eval(quote(2 + 2)))))))
   ```

2. `subset2()` has a bug if you use it with a single column data frame. What should the following code return? How can you modify `subset2()` so it returns the correct type of object?

   ```
   sample_df2 <- data.frame(x = 1:10)
   subset2(sample_df2, x > 8)
   ```

   ```
   ## Error in eval(expr, envir, enclos): could not find function "subset2"
   ```

3. The real subset function (`subset.data.frame()`) removes missing values in the condition. Modify `subset2()` to do the same: drop the offending rows.

4. What happens if you use `quote()` instead of `substitute()` inside of `subset2()`?

5. The second argument in `subset()` allows you to select variables. It treats variable names as if they were positions. This allows you to do things like `subset(mtcars, , -cyl)` to drop the cylinder variable, or `subset(mtcars, , disp:drat)` to select all the variables between `disp` and `drat`. How does this work? I've made this easier to understand by extracting it out into its own function.

```
select <- function(df, vars) {
  vars <- substitute(vars)
  var_pos <- setNames(as.list(seq_along(df)), names(df))
  pos <- eval(vars, var_pos)
  df[, pos, drop = FALSE]
}
select(mtcars, -cyl)
```

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

Exercises

1. `plyr::arrange()` works similarly to `subset()`, but instead of selecting rows, it reorders them. How does it work? What does `substitute(order(...))` do? Create a function that does only that and experiment with it.

2. What does `transform()` do? Read the documentation. How does it work? Read the source code for `transform.data.frame()`. What does `substitute(list(...))` do?

3. `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)
```

   How does mutate work? What's the key difference between `mutate()` and `transform()`?

4. What does `with()` do? How does it work? Read the source code for `with.default()`. What does `within()` do? How does it work? Read the source code for `within.data.frame()`. Why is the code so much more complex than `with()`?

Exercises

1. The following R functions all use NSE. For each, describe how it uses NSE, and read the documentation to determine its escape hatch.

    - `rm()`
    - `library()` and `require()`
    - `substitute()`
    - `data()`
    - `data.frame()`

2. Base functions `match.fun()`, `page()`, and `ls()` all try to automatically determine whether you want standard or non-standard evaluation. Each uses a different approach. Figure out the essence of each approach then compare and contrast.

3. Add an escape hatch to `plyr::mutate()` by splitting it into two functions. One function should capture the unevaluated inputs. The other should take a data frame and list of expressions and perform the computation.

4. What's the escape hatch for `ggplot2::aes()`? What about `plyr::()`? What do they have in common? What are the advantages and disadvantages of their differences?

5. The version of `subset2_q()` I presented is a simplification of real code. Why is the following version better?

    ```
    subset2_q <- function(x, cond, env = parent.frame()) {
      r <- eval(cond, x, env)
      x[r, ]
    }
    ```

    Rewrite `subset2()` and `subscramble()` to use this improved version.

Exercises

1. Use `subs()` to convert the LHS to the RHS for each of the following pairs:

    - `a + b + c` -> `a * b * c`
    - `f(g(a, b), c)` -> `(a + b) * c`
    - `f(a < b, c, d)` -> `if (a < b) c else d`

2. For each of the following pairs of expressions, describe why you can't use `subs()` to convert one to the other.

    - `a + b + c` -> `a + b * c`
    - `f(a, b)` -> `f(a, b, c)`
    - `f(a, b, c)` -> `f(a, b)`

3. How does `pryr::named_dots()` work? Read the source.

Exercises

1. What does the following function do? What's the escape hatch? Do you think that this is an appropriate use of NSE?

```
nl <- function(...) {
  dots <- named_dots(...)
  lapply(dots, eval, parent.frame())
}
```

2. Instead of relying on promises, you can use formulas created with `~` to explicitly capture an expression and its environment. What are the advantages and disadvantages of making quoting explicit? How does it impact referential transparency?

3. Read the standard non-standard evaluation rules found at http://developer.r-project.org/nonstandard-eval.pdf.

Exercises from extras/local.Rmd

Exercises

1. `local()` is hard to understand because it is very concise and uses some sutble features of evaluation (including non-standard evaluation of both arguments). Confirm that the following function works the same as `local()` and then explain how it works.

```
local3 <- function(expr, envir = new.env()) {
  call <- substitute(eval(quote(expr), envir))
  env <- parent.frame()

  eval(call, env)
}
```

Exercises from Expressions.rmd

Exercises

1. There's no existing base function that checks if an element is a valid component of an expression (i.e., it's a constant, name, call, or pairlist). Implement one by guessing the names of the “is” functions for calls, names, and pairlists.

2. `pryr::ast()` uses non-standard evaluation. What's its escape hatch to standard evaluation?

3. What does the call tree of an if statement with multiple else conditions look like?

4. Compare `ast(x + y %+% z)` to `ast(x ^ y %+% z)`. What do they tell you about the precedence of custom infix functions?

5. Why can't an expression contain an atomic vector of length greater than one? Which one of

the six types of atomic vector can't appear in an expression? Why?

Exercises

1. You can use `formals()` to both get and set the arguments of a function. Use `formals()` to modify the following function so that the default value of `x` is missing and `y` is 10.

```
g <- function(x = 20, y) {
  x + y
}
```

2. Write an equivalent to `get()` using `as.name()` and `eval()`. Write an equivalent to `assign()` using `as.name()`, `substitute()`, and `eval()`. (Don't worry about the multiple ways of choosing an environment; assume that the user supplies it explicitly.)

Exercises

1. 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
```

   What's the difference? Which one should you prefer?

2. Implement a pure R version of `do.call()`.

3. Concatenating a call and an expression with `c()` creates a list. Implement `concat()` so that the following code works to combine a call and an additional argument.

```
concat(quote(f), a = 1, b = quote(mean(a)))
#> f(a = 1, b = mean(a))
```

4. Since `list()`s don't belong in expressions, we could create a more convenient call constructor that automatically combines lists into the arguments. Implement `make_call()`

so that the following code works.

```
make_call(quote(mean), list(quote(x), na.rm = TRUE))
#> mean(x, na.rm = TRUE)
make_call(quote(mean), quote(x), na.rm = TRUE)
#> mean(x, na.rm = TRUE)
```

5. How does `mode<-` work? How does it use `call()`?

6. Read the source for `pryr::standardise_call()`. How does it work? Why is `is.primitive()` needed?

7. `standardise_call()` doesn't work so well for the following calls. Why?

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

```
## Error in eval(expr, envir, enclos): could not find function "standard
```

```
standardise_call(quote(mean(n = T, 1:10)))
```

```
## Error in eval(expr, envir, enclos): could not find function "standard
```

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

```
## Error in eval(expr, envir, enclos): could not find function "standard
```

8. Read the documentation for `pryr::modify_call()`. How do you think it works? Read the source code.

9. Use `ast()` and experimentation to figure out the three arguments in an `if()` call. Which components are required? What are the arguments to the `for()` and `while()` calls?

Exercises

1. Compare and contrast `update_model()` with `update.default()`.

2. Why doesn't `write.csv(mtcars, "mtcars.csv", row = FALSE)` work? What property of argument matching has the original author forgotten?

3. Rewrite `update.formula()` to use R code instead of C code.

4. Sometimes it's necessary to uncover the function that called the function that called the current function (i.e., the grandparent, not the parent). How can you use `sys.call()` or `match.call()` to find this function?

Exercises

1. How are `alist(a)` and `alist(a = )` different? Think about both the input and the output.

2. Read the documentation and source code for `pryr::partial()`. What does it do? How does it work? Read the documentation and source code for `pryr::unenclose()`. What does it do and how does it work?

3. The actual implementation of `curve()` looks more like

```
curve3 <- 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)))
}
```

   How does this approach differ from `curve2()` defined above?

Exercises

1. What are the differences between `quote()` and `expression()`?

2. Read the help for `deparse()` and construct a call that `deparse()` and `parse()` do not operate symmetrically on.

3. Compare and contrast `source()` and `sys.source()`.

4. Modify `simple_source()` so it returns the result of *every* expression, not just the last one.

5. The code generated by `simple_source()` lacks source references. Read the source code for `sys.source()` and the help for `srcfilecopy()`, then modify `simple_source()` to preserve source references. You can test your code by sourcing a function that contains a comment. If successful, when you look at the function, you'll see the comment and not just the source code.

Exercises

1. Why does `logical_abbr()` use a for loop instead of a functional like `lapply()`?

2. `logical_abbr()` works when given quoted objects, but doesn't work when given an existing function, as in the example below. Why not? How could you modify `logical_abbr()` to work with functions? Think about what components make up a function.

```
f <- function(x = TRUE) {
  g(x + T)
}
logical_abbr(f)
```

3. Write a function called `ast_type()` that returns either “constant”, “name”, “call”, or “pairlist”. Rewrite `logical_abbr()`, `find_assign()`, and `bquote2()` to use this function with `switch()` instead of nested if statements.

4. Write a function that extracts all calls to a function. Compare your function to `pryr::fun_calls()`.

5. Write a wrapper around `bquote2()` that does non-standard evaluation so that you don't need to explicitly `quote()` the input.

6. Compare `bquote2()` to `bquote()`. There is a subtle bug in `bquote()`: it won't replace calls to functions with no arguments. Why?

```
bquote(.(x)(), list(x = quote(f)))
```

```
## .(x)()
```

```
bquote(.(x)(1), list(x = quote(f)))
```

```
## f(1)
```

7. Improve the base `recurse_call()` template to also work with lists of functions and expressions (e.g., as from `parse(path_to_file)`).

Exercises from dsl.rmd

Exercises

1. The escaping rules for `<script>` and `<style>` tags are different: you don't want to escape angle brackets or ampersands, but you do want to escape `</script>` or `</style>`. Adapt the code above to follow these rules.

2. The use of `...` for all functions has some big downsides. There's no input validation and there will be little information in the documentation or autocomplete about how they are used in the function. Create a new function that, when given a named list of tags and their attribute names (like below), creates functions which address this problem.

```
list(
  a = c("href"),
  img = c("src", "width", "height")
)
```

All tags should get `class` and `id` attributes.

3. Currently the HTML doesn't look terribly pretty, and it's hard to see the structure. How could you adapt `tag()` to do indenting and formatting?

Exercises

1. Add escaping. The special symbols that should be escaped by adding a backslash in front of them are `\`, `$`, and `%`. Just as with HTML, you'll need to make sure you don't end up double-escaping. So you'll need to create a small S3 class and then use that in function operators. That will also allow you to embed arbitrary LaTeX if needed.

2. Complete the DSL to support all the functions that `plotmath` supports.

3. There's a repeating pattern in `latex_env()`: we take a character vector, do something to each piece, convert it to a list, and then convert the list to an environment. Write a function that automates this task, and then rewrite `latex_env()`.

4. Study the source code for `dplyr`. An important part of its structure is `partial_eval()` which helps manage expressions when some of the components refer to variables in the database while others refer to local R objects. Note that you could use very similar ideas if you needed to translate small R expressions into other languages, like JavaScript or Python.

Exercises from Performance.rmd

Exercises

1. Instead of using `microbenchmark()`, you could use the built-in function `system.time()`. But `system.time()` is much less precise, so you'll need to repeat each operation many times with a loop, and then divide to find the average time of each operation, as in the code below.

```
n <- 1:1e6
system.time(for (i in n) sqrt(x)) / length(n)
system.time(for (i in n) x ^ 0.5) / length(n)
```

How do the estimates from `system.time()` compare to those from `microbenchmark()`? Why are they different?

2. Here are two other ways to compute the square root of a vector. Which do you think will be

fastest? Which will be slowest? Use microbenchmarking to test your answers.

```
x ^ (1 / 2)
exp(log(x) / 2)
```

3. Use microbenchmarking to rank the basic arithmetic operators (`+`, `-`, `*`, `/`, and `^`) in terms of their speed. Visualise the results. Compare the speed of arithmetic on integers vs. doubles.

4. You can change the units in which the microbenchmark results are expressed with the `unit` parameter. Use `unit = "eps"` to show the number of evaluations needed to take 1 second. Repeat the benchmarks above with the eps unit. How does this change your intuition for performance?

Exercises

1. `scan()` has the most arguments (21) of any base function. About how much time does it take to make 21 promises each time scan is called? Given a simple input (e.g., `scan(text = "1 2 3", quiet = T)`) what proportion of the total run time is due to creating those promises?

2. Read "Evaluating the Design of the R Language". What other aspects of the R-language slow it down? Construct microbenchmarks to illustrate.

3. How does the performance of S3 method dispatch change with the length of the class vector? How does performance of S4 method dispatch change with number of superclasses? How about RC?

4. What is the cost of multiple inheritance and multiple dispatch on S4 method dispatch?

5. Why is the cost of name lookup less for functions in the base package?

Exercises

1. The performance characteristics of `squish_ife()`, `squish_p()`, and `squish_in_place()` vary considerably with the size of `x`. Explore the differences. Which sizes lead to the biggest and smallest differences?

2. Compare the performance costs of extracting an element from a list, a column from a matrix, and a column from a data frame. Do the same for rows.

Exercises from Profiling.rmd

Exercises

1. What are faster alternatives to `lm`? Which are specifically designed to work with larger datasets?

2. What package implements a version of `match()` that's faster for repeated lookups? How much faster is it?

3. List four functions (not just those in base R) that convert a string into a date time object. What are their strengths and weaknesses?

4. How many different ways can you compute a 1d density estimate in R?

5. Which packages provide the ability to compute a rolling mean?

6. What are the alternatives to `optim()`?

Exercises

1. How do the results change if you compare `mean()` and `mean.default()` on 10,000 observations, rather than on 100?

2. The following code provides an alternative implementation of `rowSums()`. Why is it faster for this input?

```
rowSums2 <- function(df) {
  out <- df[[1L]]
  if (ncol(df) == 1) return(out)

  for (i in 2:ncol(df)) {
    out <- out + df[[i]]
  }
  out
}

df <- as.data.frame(
  replicate(1e3, sample(100, 1e4, replace = TRUE))
)
system.time(rowSums(df))
```

```
##    user  system elapsed
##    0.09    0.00    0.09
```

```
system.time(rowSums2(df))
```

```
##    user  system elapsed
##    0.03    0.02    0.05
```

3. What's the difference between `rowSums()` and `.rowSums()`?

4. Make a faster version of `chisq.test()` that only computes the chi-square test statistic

when the input is two numeric vectors with no missing values. You can try simplifying `chisq.test()` or by coding from the mathematical definition.

5. Can you make a faster version of `table()` for the case of an input of two integer vectors with no missing values? Can you use it to speed up your chi-square test?

6. Imagine you want to compute the bootstrap distribution of a sample correlation using `cor_df()` and the data in the example below. Given that you want to run this many times, how can you make this code faster? (Hint: the function has three components that you can speed up.)

```
n <- 1e6
df <- data.frame(a = rnorm(n), b = rnorm(n))

cor_df <- function(i) {
  i <- sample(seq(n), n * 0.01)
  cor(q[i, , drop = FALSE])[2,1]
}
```

Is there a way to vectorise this procedure?

Exercises

1. The density functions, e.g., `dnorm()`, have a common interface. Which arguments are vectorised over? What does `rnorm(10, mean = 10:1)` do?

2. Compare the speed of `apply(x, 1, sum)` with `rowSums(x)` for varying sizes of `x`.

3. How can you use `crossprod()` to compute a weighted sum? How much faster is it than the naive `sum(x * w)`?

Exercises from memory.rmd

Exercises

1. Repeat the analysis above for numeric, logical, and complex vectors.

2. If a data frame has one million rows, and three variables (two numeric, and one integer), how much space will it take up? Work it out from theory, then verify your work by creating a data frame and measuring its size.

3. Compare the sizes of the elements in the following two lists. Each contains basically the same data, but one contains vectors of small strings while the other contains a single long string.

```
vec <- lapply(0:50, function(i) c("ba", rep("na", i)))
str <- lapply(vec, paste0, collapse = "")
```

4. Which takes up more memory: a factor (`x`) or the equivalent character vector (`as.character(x)`)? Why?

5. Explain the difference in size between `1:5` and `list(1:5)`.

Exercises

1. When the input is a list, we can make a more efficient `as.data.frame()` by using special knowledge. A data frame is a list with class `data.frame` and `row.names` attribute. `row.names` is either a character vector or vector of sequential integers, stored in a special format created by `.set_row_names()`. This leads to an alternative `as.data.frame()`:

   ```
   to_df <- function(x) {
     class(x) <- "data.frame"
     attr(x, "row.names") <- .set_row_names(length(x[[1]]))
     x
   }
   ```

   What impact does this function have on `read_delim()`? What are the downsides of this function?

2. Line profile the following function with `torture = TRUE`. What is surprising? Read the source code of `rm()` to figure out what's going on.

   ```
   f <- function(n = 1e5) {
     x <- rep(1, n)
     rm(x)
   }
   ```

Exercises

1. The code below makes one duplication. Where does it occur and why? (Hint: look at `refs(y)`.)

   ```
   y <- as.list(x)
   ```

   ```
   ## Error in as.list(x): object 'x' not found
   ```

   ```
   for(i in seq_along(medians)) {
     y[[i]] <- y[[i]] - medians[i]
   }
   ```

   ```
   ## Error in eval(expr, envir, enclos): object 'medians' not found
   ```

2. The implementation of `as.data.frame()` in the previous section has one big downside. What is it and how could you avoid it?

Exercises from Rcpp.rmd

Exercises

With the basics of C++ in hand, it's now a great time to practice by reading and writing some simple C++ functions. For each of the following functions, read the code and figure out what the corresponding base R function is. You might not understand every part of the code yet, but you should be able to figure out the basics of what the function does.

```
double f1(NumericVector x) {
  int n = x.size();
  double y = 0;

  for(int i = 0; i < n; ++i) {
    y += x[i] / n;
  }
  return y;
}

NumericVector f2(NumericVector x) {
  int n = x.size();
  NumericVector out(n);

  out[0] = x[0];
  for(int i = 1; i < n; ++i) {
    out[i] = out[i - 1] + x[i];
  }
  return out;
}

bool f3(LogicalVector x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    if (x[i]) return true;
  }
  return false;
}

int f4(Function pred, List x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    LogicalVector res = pred(x[i]);
    if (res[0]) return i + 1;
  }
  return 0;
}

NumericVector f5(NumericVector x, NumericVector y) {
  int n = std::max(x.size(), y.size());
  NumericVector x1 = rep_len(x, n);
  NumericVector y1 = rep_len(y, n);

  NumericVector out(n);
```

```
  for (int i = 0; i < n; ++i) {
    out[i] = std::min(x1[i], y1[i]);
  }

  return out;
}
```

To practice your function writing skills, convert the following functions into C++. For now, assume the inputs have no missing values.

1. `all()`
2. `cumprod()`, `cummin()`, `cummax()`.
3. `diff()`. Start by assuming lag 1, and then generalise for lag `n`.
4. `range`.
5. `var`. Read about the approaches you can take on wikipedia. Whenever implementing a numerical algorithm, it's always good to check what is already known about the problem.

Exercises

1. Rewrite any of the functions from the first exercise to deal with missing values. If `na.rm` is true, ignore the missing values. If `na.rm` is false, return a missing value if the input contains any missing values. Some good functions to practice with are `min()`, `max()`, `range()`, `mean()`, and `var()`.
2. Rewrite `cumsum()` and `diff()` so they can handle missing values. Note that these functions have slightly more complicated behaviour.

Exercises

To practice using the STL algorithms and data structures, implement the following using R functions in C++, using the hints provided:

1. `median.default()` using `partial_sort`.
2. `%in%` using `unordered_set` and the `find()` or `count()` methods.
3. `unique()` using an `unordered_set` (challenge: do it in one line!).
4. `min()` using `std::min()`, or `max()` using `std::max()`.
5. `which.min()` using `min_element`, or `which.max()` using `max_element`.
6. `setdiff()`, `union()`, and `intersect()` for integers using sorted ranges and `set_union`, `set_intersection` and `set_difference`.