MATH 3080 R Lecture Notes

Curtis Miller

2020-02-14

Contents

Preface

These lecture notes were written by me to accompany John Verzani's *Using R for Introductory Statistics (2nd ed.)*, to be delivered in lectures teaching students how to program with R in the programming lab accompanying a lecture section focusing on the statistical methods themselves.

These notes are not intended to stand alone; I like Verzani's book and I believe that these notes should supplement it, not replace it. For those taking the programming lab for the University of Utah's Mathematics Department statistics courses, I would *insist* on reading Verzani's book in addition to these lecture notes. However, these notes could serve as a light weight introduction to R and statistical programming.

In any case, I hope that you find these notes useful, and wish you the best of luck.

Curtis Miller

Lecture 1

Control Flow

In programming, **control flow** is the order in which statements in a program are evaluated, if they're evaluated at all. R is a full-featured programming language and thus allows for control flow statements. We will review those statements here.

`if` and `else`

`if` statements allows for conditional evaluation of code, and take the form `if (condition) {code}`.[1] `condition` must be a statement that evaluates to a single `TRUE` or `FALSE`. If `condition` is `TRUE`, then `code` will be run. However, programmers may want certain code to run if `condition` is `FALSE`; for this purpose, `else` exists, and we may use it in the format `if (condition) {code_true} else {code_false}`. In fact, programmers may want to check a series of possibilities and execute code based on which is true, in which case we can write `else if` to add more contingencies to our `if` statement.

Below are some examples of using `if` statements.

```r
if (1 + 1 == 2) {
    print("Arithmetic works!")
}
```

```
## [1] "Arithmetic works!"
```

```r
if (0 < 1) {
    print("We have order!")
} else {
    print("We don't have order!")
}
```

```
## [1] "We have order!"
```

---

[1] When on a single line, the braces `{}` are actually optional, but I recommend always using them for style purposes; your code is more robust and easily changed if you always use braces.

```
x <- 0

if (x < 0) {
    print("x is less than zero")
} else if (x == 0) {
    print("x is zero")
} else {
    print("x is greater than zero")
}
```

```
## [1] "x is zero"
```

Similar to `if` and `else` is the `ifelse()` function, used like so: `ifelse(vec, return_true, return_false)`. `ifelse()` is a vectorized function; `vec` could be a vector consisting of `TRUE` and `FALSE` value. For every `TRUE` in `vec`, `return_true` will be returned; elsewhere, `return_false` is returned.

```
(x <- rnorm(10))
```

```
##  [1] -0.8820750 -0.4256935  0.6602596 -1.8273232 -1.0539485  1.3842224
##  [7]  0.8441204  0.3333570  0.7442229 -1.1100196
```

```
# We will return a vector where all negative numbers are zero; otherwise, the
# original value of the vector is kept.

x < 0
```

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

```
ifelse(x < 0, 0, x)  # Put 0 where TRUE, else put corresponding x value
```

```
##  [1] 0.0000000 0.0000000 0.6602596 0.0000000 0.0000000 1.3842224 0.8441204
##  [8] 0.3333570 0.7442229 0.0000000
```

`switch()`

Suppose you wanted a value to depend on some input, and many different values for that input are possible. For example, a string determines the statistic to return. We could chain `if` and `else` statements like so:

```
stat <- "mean"
x <- rnorm(10)

if (stat == "mean") {
    mean(x)
} else if (stat == "median") {
    median(x)
} else if (stat == "max") {
    max(x)
} else if (stat == "min") {
    min(x)
}
```

```
## [1] -0.3233816
```

The above code works but is verbose and error-prone[2]. We could tidy our code by using `switch()` which allows for passing parameters that can provide multiple outputs for multiple cases.

Here's the above code block tidied by `switch()`:

```
switch(stat,
  mean = mean(x),
  median = median(x),
  max = max(x),
  min = min(x)
)
```

```
## [1] -0.3233816
```

`tryCatch()`

`tryCatch()` is a function that allows for conditionally running code depending on whether an error appeared or not. Rather than a program simply failing when an error occurs, `tryCatch()` facilitates nuanced approaches to error handling. A common syntax for `tryCatch()` is `tryCatch(expr, error = function(e) {do_something()})`. `expr` is some code to be run, and the function passed to the `error` argument handles the error object `e`. An optional parameter, `finally`, allows for an expression that will always be run regardless of whether there was an error or not, and will be evaluated before the result of `expr` is returned.

```
tryCatch(1 + 1, error = function(e) {"Help!"}, finally = print("Hello!"))
```

```
## [1] "Hello!"
```

```
## [1] 2
```

```
tryCatch(1 + "a", error = function(e) {"Help!"}, finally = print("Hello!"))
```

```
## [1] "Hello!"
```

```
## [1] "Help!"
```

```
# Errors are actually objects that we can inspect; here I capture this object
error_obj <- tryCatch(1 + "a", error = function(e) {e})
str(error_obj)
```

---

[2] The above code could be quickly written and vastly improved like so: `stat <- get("mean"); stat(x)`. Try it! What just happened? What does `get()` do? Without using `get()` or a string, we could similarly try `stat <- mean; stat(x)`. Either solution is probably better than using `switch()`, though.

```
## List of 2
##  $ message: chr "non-numeric argument to binary operator"
##  $ call   : language 1 + "a"
##  - attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

```
error_obj$message
```

```
## [1] "non-numeric argument to binary operator"
```

Repeated Evaluation

A **loop** is a section of code evaluated repeatedly before a program proceeds to later code. All loops consist of a body of code to be repeated and a condition determining if the loop needs to be terminated. **Beware:** if this condition never occurs, the loop will never end and the program will never stop unless some outside force (such as a kill signal from the operating system) terminates the program.[3]

A `while` loop may be the simplest loop; the body is run until the condition passed to the loop becomes false. These loops take the form `while (condition) {code}`.

```
x <- 0
while (x < 10) {
    print(x)
    x <- x + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

Below is a simple loop that never ends:

```
while (TRUE) {
    print("I'm a potatoe!")
}
```

---

[3] Actually many programs consist of loops that basically never end. Video games and graphical programs, for example, are understood as operating in a loop that will not terminate until the whole program terminates.

`for` loops use list-like objects in their condition and also have a variable that represents each element in the list. The syntax for `for` loops is `for (var in l) {do_something_with(var)}`. These loops can be viewed as `while` loops where the condition for continuation is that there are more elements in the list to process.

```
x <- 1:10
for (i in x) {
    print(i)
}
```

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

```
# A Fibonacci number calculator
m <- 10  # Tenth number in Fibonacci sequence
f_num <- 1  # The first number in the sequence
s_num <- 1  # The second number in the sequence
for (i in 1:(m - 2)) {
  t_num <- f_num + s_num
  f_num <- s_num
  s_num <- t_num
}
t_num
```

```
## [1] 55
```

```
# Equivalent while loop
f_num <- 1
s_num <- 1
l <- 1:(m - 2)
n <- length(l)
idx <- 1
while (idx <= n) {
  i <- l[idx]
  t_num <- f_num + s_num
  f_num <- s_num
  s_num <- t_num
  idx <- idx + 1
}
```

```
t_num
```

```
## [1] 55
```

The expression `break` when used in a loop allows for early termination of the loop.

```
x <- 0
while (TRUE) {  # Will this loop end?
    x <- x + 1
    if (x > 10) {
        print("Potatoe!")
        break
    }
}
```

```
## [1] "Potatoe!"
```

`repeat` is essentially a `while` loop with a condition that's always true; the only way to break the loop (other than a kill signal) is via `break`.

```
# This loop is equivalent to the loop above
x <- 0
repeat {
  x <- x + 1
  if (x > 10) {
    print("Potatoe!")
    break
  }
}
```

```
## [1] "Potatoe!"
```

You may have heard never to use loops such as `for` loops when programming in R. There are good reasons to avoid `for` loops; you should not use them in place of vectorized operations, for example. However, there are times when loops are unavoidable (the Fibonacci sequence is an example of a sequence that's hard to handle without loops) and when programming one should be ready to use any necessary loops.

Throwing Errors and Warnings

While novice programmers strongly dislike errors they are important for programming. Ample errors and warnings allow for easier debugging and prevent programs from entering unwanted territory; an error thrown near the initial bad code can save hours of time trying to find the initial problem. Thus programmers look for places to throw errors and warnings to make sure that programs run as expected and don't behave badly.

In R there are three classes of run-time messaging that can be used for behavior management; ranked from most to least severe, there are **errors**, **warnings**, and **messages**. We can throw an error using the function `stop()`, where the input to `stop()` is the message to accompany the error.

```r
x <- "a"
if (is.character(x)) {
  stop("x should not be a character")
} else {
  x + 1
}
```

```
## Error in eval(expr, envir, enclos): x should not be a character
```

Errors should be thrown via `stop()` when the appropriate course of action is for the program to stop and not proceed. This contrasts with throwing a warning, where the program can still proceed but the user should be notified that some belief about the state of the program has been violated.

```r
u <- 1
if (!is.logical(u)) {
  warning("u is not boolean!")
}
```

```
## Warning: u is not boolean!
```

```r
u | FALSE
```

```
## [1] TRUE
```

Finally there are messages. Messages simply alert the user to the current operation of the program; they do not necessarily indicate "bad" behavior. Packages, for example, produce messages as they load.

```r
sum <- 0
for (i in 1:10) {
  message(i)
  sum <- sum + i
}
```

```
## 1

## 2

## 3

## 4

## 5

## 6

## 7
## 8
```

```
## 9
```

```
## 10
```

```
sum
```

```
## [1] 55
```

Functions

You should already have seen functions and function authoring before, so consider this sentence review: a **function** is a structure in a program that will execute some segment of code when called. Functions often take inputs and will return outputs depending on those inputs. In R, functions are objects that can be created like any other data object; to repeat, functions in R can be treated as data. This means that:

- You can save function in variables, vectors, lists, etc. (For example, the following is legal and perfectly reasonable code: `c(mean, median, sd)`.)
- Functions can be passed to functions as arguments; we call functions accepting functions as inputs **functionals**. (An example of functionals are the `apply()` collection of functions, including `lapply()`.)
- Functions can be *returned* by functions, as an output; we call the function returned by another function a **closure**. (An example of a function that produces closures is `Vectorize()`.)

Functions consist of three key ingredients: the **formals**, the **body**, and the **environment**. Formals are arguments the function accepts. The function body is the block of code executed by the function when called. The environment is the data structure in which the function is defined; user functions generally are defined in the global environment, but closures often live in a different environment.

The following illustration demonstrates the relationship between these three things:

```
# This function was defined by the user and thus lives in the global environment

function(formals) {
  body
}
```

There are functions that pick a function apart into these component parts:

```
increment <- function(x) {
    x + 1
}
formals(increment)
```

```
## $x
```

```
body(increment)
```

```
## {
##     x + 1
## }
```

```
environment(increment)
```

```
## <environment: R_GlobalEnv>
```

The function `args()` is primarily interested in the formals of a function, but for interactive use, showing R users what arguments a function takes and their default values. `args()` should only be used interactively to personally learn about a funciton, not for more advanced programming; when programming, use `formals()`.

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL)
## NULL
```

```
formals(paste)  # The result is a list with named elements, and entries of the
```

```
## $...
##
##
## $sep
## [1] " "
##
## $collapse
## NULL
```

```
                # list being parameter defaults
```

Infix Notation

Recall the rules for function names; syntactically valid names include alphanumeric characters, `.`, and `_`, but cannot start with numbers or `_` (so `.Mean_1_` is a valid name, but `1.Mean_` and `_Mean.1_` are not). However, we can give objects syntactically invalid names by using backquotes, like so:

```
.Mean_1_ <- 1           # Syntactically valid name
`1.Mean_` <- 2          # Invalid name
`_Mean.1_` <- 3         # Invalid name
`Awesome sauce!` <- 4   # Invalid name

.Mean_1_
```

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

```r
`.Mean_1_`
```

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

```r
`1.Mean_`
```

```
## [1] 2
```

```r
`_Mean.1_`
```

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

```r
`Awesome sauce!`
```

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

One particular case where we may want syntactically invalid names is to define infix functions. You've already met one: `+`. Yes, `+` is a function, as demonstrated below:

```r
`+`
```

```
## function (e1, e2)  .Primitive("+")
```

```r
class(`+`)
```

```
## [1] "function"
```

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

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

In short, an infix function is a function `f` that can be called with two arguments like so: `x f y`. Most user-defined infix functions, though, need to have their name wrapped by two `%` symbols, like `%my_function%`.

R does come with some such function already defined (excluding "trivial" ones like `+`). `%*%` computes matrix (inner) products (the product of two matrices as taught in linear algebra, as opposed to element-wise products as done by `*`), and `%o%` matrix outer products. The **dplyr** operator `%>%` is another example. These are not the only useful infix operators we could imagine, though.

For example, R does not have an operator for string concatenation (meaning combining two strings; when we concatenate `"string1"` and `"string2"`, we get the string `"string1string2"`). We can define an infix operator for string concatenation like so:

```r
`%s%` <- function(x, y) {paste(x, y)}    # Concatenate, separating with space
`%s0%` <- function(x, y) {paste0(x, y)}  # Concatenate, no separating characters

"hello" %s% "world"
```

```
## [1] "hello world"
```

```
"hello" %s0% "world"
```

```
## [1] "helloworld"
```

Variadic Arguments

When looking at some function arguments, you may on occasion notice the argument `...`. This argument actually refers to a list of arguments of undefined length. This allows for writing functions that can take arguments not necessarily defined outright by the programmer in the function definition.

We allow for variadic arguments when we include `...` in the function definition, and use them by referring to `...` in a function call. Any argument passed to the function that was not named as an argument to the function is included in `...`.

Below is a function that collects some arguments in `...` and attempts to return them in a vector (this could be problematic if not all the arguments are of the same type; I would recommend collecting in a list instead in general). The function also has a named argument that gets printed.

```
collector <- function(..., stringout = "Hello!") {
    print(stringout)
    vec <- c(...)
    return(vec)
}

u <- collector(1, 1, 4)
```

```
## [1] "Hello!"
```

```
u
```

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

```
u <- collector(a = 1, b = 1, c = 1, stringout = 10)
```

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

```
u
```

```
## a b c
## 1 1 1
```

One can place `...` anywhere in the funciton defintion, but beware: where it's placed matters. Named arguments prior to `...` are treated as positional arguments, and arguments after `...` must be referred to explicitly if they are to be modified.

```
new_collector <- function(introstring = "Hello!", ...,
                          leavestring = "Goodbye!") {
  print(introstring)
```

```r
  u <- c(...)
  print(leavestring)

  u  # u will be returned when referred to like this
}

x <- new_collector("Whoa", "Awesome", "world")
```

```
## [1] "Whoa"
## [1] "Goodbye!"
```

```r
x
```

```
## [1] "Awesome" "world"
```

```r
x <- new_collector(1, 2, 3, 4)
```

```
## [1] 1
## [1] "Goodbye!"
```

```r
x
```

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

```r
x <- new_collector("whoa", "awesome", "stuff", leavestring = "solo")
```

```
## [1] "whoa"
## [1] "solo"
```

```r
x
```

```
## [1] "awesome" "stuff"
```

```r
x <- new_collector(1, 2, 3, introstring = "whoa", leavestring = "awesome")
```

```
## [1] "whoa"
## [1] "awesome"
```

```r
x
```

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

So far I have demonstrated `...` by immediately passing it to `c()`, but in fact we can pass `...` to any function. Let's demonstrate by writing our own version of the function `paste()`. The argument `sep` controls the separator between words in the string. We could set `sep = "_"` to separate with _ rather than a space (the default). That said, the *only* parameter we want to change is `sep`; we otherwise want our new function to behave exactly like `paste()`, and we don't want to have to rewrite `paste()` to accomodate this. Also, we prefer simple code; we don't want to have to repeat all of `paste()`'s arguments in our function definition, especially since we want our new function to work with all past and future versions of `paste()` that could behave differently in other versions of R.

(Actually I don't expect the `paste()` syntax to ever change, but the moral of the story stands.)

Given these design constraints, `...` is extremely useful, since we only need to write the following:

```r
paste_ <- function(..., sep = "_") {
    paste(..., sep = sep)
}
paste("hello", "world")
```

```
## [1] "hello world"
```

```r
paste_("hello", "world")
```

```
## [1] "hello_world"
```

```r
paste_("source", 1:10)
```

```
##  [1] "source_1"  "source_2"  "source_3"  "source_4"  "source_5"
##  [6] "source_6"  "source_7"  "source_8"  "source_9"  "source_10"
```

```r
paste_("source", 1:10, collapse = "...")             # A paste() argument
```

```
## [1] "source_1...source_2...source_3...source_4...source_5...source_6...source_7...source_8...s
```

```r
paste_("source", 1:10, collapse = "...", sep = "-")  # Stupid but valid
```

```
## [1] "source-1...source-2...source-3...source-4...source-5...source-6...source-7...source-8...s
```

Demonstration: z Statistic Function

Recall from previous courses the z statistic:

$$z = \frac{\bar{x} - \mu_0}{\frac{\sigma}{\sqrt{n}}}$$

Here, $\bar{x}$ is the sample mean, n the sample size, μ_0 is the mean of the data under the null hypothesis, and σ is the population standard deviation. Let's write a function that computes the z statistic. How will our function work? We will sketch the following specification:

- We will call our function `z.stat()`.
- `z.stat()` will take a list of numbers *directly* and compute the z statistic. (This is probably not wise from a design perspective but allows us to demonstrate the use of `...`.)
- Two named parameters for `z.stat()` will be `sigma` and `mu`, representing the population standard deviation and mean under the null hypothesis, respectively

- The function should raise an error if the input data is not numeric or if `sigma` is not positive. We want the function to check and throw its own errors in these cases; while we might consider letting the arithmetic throw the errors, having the function throw the error allows for easier error diagnosis should an unwitting user make a mistake.
- We want the function to work for boolean (i.e. `TRUE`/`FALSE`) data but throw a warning; while boolean can be easily converted to numeric, we don't want our function used this way, though we don't want to explicitly ban this usage.
- The user does not absolutely have to specify `sigma`; our function could fall back on the sample standard deviation in that case. However, we don't want to encourage this use, so we should throw a warning if this occurs.

(I'm not going to argue this is good design; this is more about demonstrating techniques than using them well. That said, writing a design document for code is a very good idea, and essential if the code is going to be distributed or if a team is working on it.)

```r
# The following comments are more than just comments; they're ready for
# interpretation by a package called roxygen2, which can create documentation
# for functions and other package objects.

#' Z Statistic
#'
#' Compute the \eqn{z}-statistic for a given data set.
#'
#' @param ... Data for which to compute the \eqn{z}-statistic
#' @param mu The mean of the population under the null hypothesis
#' @param sigma The population standard deviation; unset by default (this is
#'              achieved by making the default value of the parameter
#'              \code{NULL})
#' @return The \eqn{z}-statistic
#' @examples
#' z.stat(1, 2, 3, mu = 2, sigma = 1)
z.stat <- function(..., mu = 0, sigma = NULL) {
  x <- tryCatch(c(...),  # Attempt to convert ... into a vector
      error = function(e) {
        "Inappropriate data passed to z.stat(); it should be numeric"
      })
  if (is.logical(x)) {
    warning("Data passed to z.stat() is logical; may be inappropriate for z" %s%
            "test")
  } else if (!is.numeric(x)) {
    stop("Data passed to z.stat() must be numeric")
  }
  if (is.null(sigma)) {
    warning("sigma not set; defaulting to sample standard deviation, but" %s%
            "statistic may not be appropriate")
    sigma <- sd(x)
  }

  if (!is.numeric(sigma) | sigma <= 0) {
    stop("sigma must be a positive number")
  }

  xbar <- mean(x)
  n <- length(x)

  (xbar - mu) / (sigma / sqrt(n))
}
```

Let's run some tests to see that our function works as intended.

```
z.stat(0.1, 1.1, -0.1, -2.5, 0.3)  # Produces a warning
```

```
## Warning in z.stat(0.1, 1.1, -0.1, -2.5, 0.3): sigma not set; defaulting to
## sample standard deviation, but statistic may not be appropriate
```

```
## [1] -0.3634502
```

```
z.stat(0.1, 1.1, -0.1, -2.5, 0.3, sigma = 1)
```

```
## [1] -0.491935
```

```
z.stat(0.1, 1.1, -0.1, -2.5, 0.3, mu = 1, sigma = 1)
```

```
## [1] -2.728003
```

```
z.stat("0.1", 1.1, -0.1, -2.5, 0.3, mu = 1, sigma = 1)  # Should produce error
```

```
## Error in z.stat("0.1", 1.1, -0.1, -2.5, 0.3, mu = 1, sigma = 1): Data passed to z.stat() must
```

```
z.stat(TRUE, TRUE, FALSE, TRUE, FALSE, mu = 1, sigma = 1)  # Produces warning
```

```
## Warning in z.stat(TRUE, TRUE, FALSE, TRUE, FALSE, mu = 1, sigma = 1): Data
## passed to z.stat() is logical; may be inappropriate for z test
```

```
## [1] -0.8944272
```

```
z.stat(0.1, 1.1, -0.1, -2.5, 0.3, mu = 1, sigma = -1)  # Produces error
```

```
## Error in z.stat(0.1, 1.1, -0.1, -2.5, 0.3, mu = 1, sigma = -1): sigma must be a positive numbe
```

Based on our tests our function seems to work appropriately.

Lecture 2

Environments

Environments are special data structures that are responsible for handling the names of variables and for looking up variables when requested. You have been interacting with environments for as long as you have been using R. Whenever you create a variable in R, what you have actually done is created an object then created a slot in the global environment (the environment where user action in R generally takes place) that points to that object. If an object has no slot in an environment pointing to that object, R automatically destroys it during a process known as garbage collection.

The function `ls()` lists the objects contained in an environment. By default, it lists what's in the global environment.

```
ls()
```

```
##  [1] "_Mean.1_"        "%s%"             "%s0%"            "1.Mean_"
##  [5] "Awesome sauce!" "collector"       "error_obj"       "f_num"
##  [9] "i"               "idx"             "increment"       "l"
## [13] "m"               "n"               "new_collector"   "paste_"
## [17] "s_num"           "stat"            "sum"             "t_num"
## [21] "u"               "x"               "z.stat"
```

```
ls(envir = globalenv())  # globalenv() accesses the global environment
```

```
##  [1] "_Mean.1_"        "%s%"             "%s0%"            "1.Mean_"
##  [5] "Awesome sauce!" "collector"       "error_obj"       "f_num"
##  [9] "i"               "idx"             "increment"       "l"
## [13] "m"               "n"               "new_collector"   "paste_"
## [17] "s_num"           "stat"            "sum"             "t_num"
## [21] "u"               "x"               "z.stat"
```

```
x <- 1
ls()
```

```
##  [1] "_Mean.1_"        "%s%"             "%s0%"            "1.Mean_"
##  [5] "Awesome sauce!" "collector"      "error_obj"      "f_num"
##  [9] "i"              "idx"            "increment"      "l"
## [13] "m"              "n"              "new_collector"  "paste_"
## [17] "s_num"          "stat"           "sum"            "t_num"
## [21] "u"              "x"              "z.stat"
```

```
ls(envir = globalenv())
```

```
##  [1] "_Mean.1_"       "%s%"            "%s0%"           "1.Mean_"
##  [5] "Awesome sauce!" "collector"      "error_obj"      "f_num"
##  [9] "i"              "idx"            "increment"      "l"
## [13] "m"              "n"              "new_collector"  "paste_"
## [17] "s_num"          "stat"           "sum"            "t_num"
## [21] "u"              "x"              "z.stat"
```

The function `environment()` lists the current environment.

```
environment()
```

```
## <environment: R_GlobalEnv>
```

In fact, we can access objects in environments similarly to how we access objects in lists. However, environments are not just a different kind of list. We'll get into that in a second; let's first see how environments have similar syntax to lists. We can create a new environment with the function `new.env()`. Objects in the environment can be referenced using `$`, like so:

```
my_env <- new.env()
my_env$x <- 1
my_env[["y"]] <- 2
my_env[["awesome sauce"]] <- 3

my_env[["x"]] + my_env$y + my_env$`awesome sauce`
```

```
## [1] 6
```

```
ls(envir = my_env)
```

```
## [1] "awesome sauce" "x"             "y"
```

However, despite these similarities, there are important distinctions between environments and lists, such as objects saved in an environment are unordered; there is no “first” item. So in the above code, `my_env[1]` and `my_env[[1]]` make no sense. Names in environments are unique, and environments have reference semantics. Additionally, you don't remove objects from environments by setting them to `NULL`, as you would with a list. Instead, you have to use the function `rm()`, like so:

```
rm(list = "awesome sauce", envir = my_env)
```

There are two fundamental ingredients needed for an environment: a **frame** (which are the name-object bindings demonstrated above) and the environment's **parent** environment, which is an environment that "contains" the environment. There is only one environment that does not have a parent: the empty environment (created automatically and referenced with `emptyenv()`). Otherwise, every environment has a parent. Aside from `emptyenv()`, two other important environments are the global environment (referenced via `globalenv()` and represents the environment that the user generally works in) and `baseenv()` (the environment of the **base** package).

When we create an environment, we can declare its parent like so:

```r
other_env <- new.env(parent = my_env)
other_env$z <- "great!"
my_env
```

```
## <environment: 0x55cdfaf8a8d8>
```

```r
parent.env(other_env)  # Find the parent of an environment
```

```
## <environment: 0x55cdfaf8a8d8>
```

By default, the parent of an environment created via `new.env()` is the global environment. When using environments as data structures (say, as a substitute for a list), consider making the parent the empty environment.

Due to the parent-child nature of environments, we could say that any environment has a sequence of ancestors, consisting of the parent of a given environment, the parent of the parent environment, and so on. The only environment that does not have a parent is the empty environment. Furthermore, the empty environment is the ultimate ancestor of all environments.

To compare environments, we use a function called `identical()`; we cannot use `==`. I demonstrate use of `identical()` below:

```r
parent.env(globalenv())
```

```
## <environment: package:stats>
## attr(,"name")
## [1] "package:stats"
## attr(,"path")
## [1] "/usr/lib/R/library/stats"
```

```r
identical(emptyenv(), globalenv())
```

```
## [1] FALSE
```

```r
identical(my_env, other_env)
```

```
## [1] FALSE
```

```r
identical(my_env, my_env)
```

```
## [1] TRUE
```

Functions and Environments

Why does this discussion matter? Users generally are not creating environments. Instead, environments are generated automatically when functions are called. Users don't notice because those environments are often instantly destroyed when the function completes and terminates. However, this matters when understanding and creating closures.

Consider the following function:

```
x <- 10
y <- 20
f <- function() {
  x <- 30
  cat("x is", x, "\n")
  cat("y is", y, "\n")
}
f()
```

```
## x is 30
## y is 20
```

```
x
```

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

```
y
```

```
## [1] 20
```

```
x <- 40
y <- 50
f()
```

```
## x is 30
## y is 50
```

```
x
```

```
## [1] 40
```

```
y
```

```
## [1] 50
```

What happened here? Here's a step-by-step breakdown of the above code sequence:

1. Variables `x` and `y` were created in the global namespace with some initial values.

2. The function `f()` was defined, then called. When `f()` was called, a new environment was temporarily created, with the global environment being its parent.
3. A variable `x` was defined in the environment where `f()` operated. This `x` is distinct from the `x` that was defined in the global environment, since they were defined in two separate environments. This is the reason why the value of `x` in the global environment was not changed.
4. When `x` was referenced in `f()`, R first looked inside the active environment, the temporary one created when the function was called. `x` was found there, so the `x` that existed in the global environment was ignored. When `y` was referenced, no `y` was found in the active environment, so R looked in the parent environment, which was the global environment. A definition for `y` was found there, and so it was the variable referenced.
5. We later changed the value of `x` and `y` in the global environment. The first temporary environment from when `f()` was called the first time was destroyed when the function finished its execution. When we called the function a second time, a brand new environment was created, with the global environment as its parent. `x` was still created in this new environment, with its same value as before. The change to `x` in the global environment didn't affect the function's execution. The reference to `y`, though, did cause different behavior, since the `y` referred to in the function was the `y` that existed in the global environment.

What if we wanted to change the value of a variable in the global environment from the function? We could do so by using `<<-`, which will only create a variable if no name is found in any of the current environment's ancestors, and if no reference is found, the variable is created in the global environment. `<<-` is demonstrated below:

```r
x <- 10
y <- 20
g <- function() {
  x <<- 30
  cat("x is", x, "\n")
  cat("y is", y, "\n")
}
g()
```

```
## x is 30
## y is 20
```

```r
x
```

```
## [1] 30
```

```r
y
```

```
## [1] 20
```

Now this time `x` was changed in the global environment. This is because when `<<-` was called, it looked for `x` in the active environment, didn't find it, then looked in the parent of the active environment, which was the global environment. It found `x` there, and modified its value. When `x` and `y` were referenced in the function, *both* of them were from the global environment (before, only `y` was a global variable).

Namespaces

A **namespace** is a special environment associated with a package. Each package has its own namespace that is attached to the global environment when the package is loaded. However, thanks to namespaces, we can even reference functions in packages without even loading the package.

We reference objects in namespaces via either `::` or `:::`, using syntax such as `namespace::object` or `namespace:::object`. The difference between `::` and `:::` is that `::` only accesses public objects, or objects that the package authors have marked as available to all of R when the package is referenced. `:::` accesses *all* objects in a package, including private ones that the package authors did not intend to make available to other code and probably don't *want* to be available. While referencing package objects via `::`, referencing via `:::` is unsafe and should be avoided.

For example, the function `mvrnorm()` from the package **MASS** allows for simulating random variables that are joinly Normal. We can use this function without explicitly loading **MASS** via `::` like so:

```
MASS::mvrnorm(n = 10, mu = c(1, 1), Sigma = matrix(c(1, 0.5, 0.5, 1), nrow = 2))
```

```
##              [,1]       [,2]
##  [1,]  0.96230907  1.1184061
##  [2,]  2.52632180  2.4991550
##  [3,]  0.18156713  1.2350943
##  [4,]  1.39059996  1.7534182
##  [5,]  0.09997212  0.3753247
##  [6,]  0.93549445 -0.1150898
##  [7,]  0.40520870  0.4681733
##  [8,]  0.54726815  1.8557975
##  [9,] -0.09466153  1.6461039
## [10,]  3.23136725  1.5636069
```

Why do namespaces matter? Consider the function `sd()` which is a function from the packages **stats** that computes standard deviations. Reading the function's code reveals that it calls the function `var()`, also from **stats**, to compute the standard deviation.

```
sd
```

```
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
## <bytecode: 0x55cdfa22dba8>
## <environment: namespace:stats>
```

The following code reveals that both `sd()` and `var()` "live" in an environment that is not the global environment.

```
environment(sd)
```

```
## <environment: namespace:stats>
```

```
environment(var)
```

```
## <environment: namespace:stats>
```

What if we change `var`? Will doing so break `sd()`?

```
x <- rnorm(10)
var(x)
```

```
## [1] 0.4825343
```

```
sd(x)
```

```
## [1] 0.6946469
```

```
var <- function(x) {10}
var(x)
```

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

```
sd(x)
```

```
## [1] 0.6946469
```

The output of `sd()` is unchanged. This is because `sd()` refers to the version of `var()` that lives in its namespace, and doesn't care about the version that lives in the global environment. The `var()` in `sd()`'s namespace can't be modified in an active R session; one would have to modify the code of the **stats** package in order to change `var()`. Therefore, `sd()` performs as expected.

If we wanted to go back to the old `var()`, we could do so via the following code:

```
var
```

```
## function(x) {10}
```

```
var <- stats::var
var
```

```
## function (x, y = NULL, na.rm = FALSE, use)
## {
##     if (missing(use))
##         use <- if (na.rm)
##             "na.or.complete"
##         else "everything"
##     na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
##         "everything", "na.or.complete"))
##     if (is.na(na.method))
##         stop("invalid 'use' argument")
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     else stopifnot(is.atomic(x))
##     if (is.data.frame(y))
##         y <- as.matrix(y)
##     else stopifnot(is.atomic(y))
##     .Call(C_cov, x, y, na.method, FALSE)
## }
## <bytecode: 0x55cdfa50ee20>
## <environment: namespace:stats>
```

Closures

The above discussions about environments may have been enlightening, but the primary motivation was to allow discussion of writing functions that can return **closures**, which are also functions (the term simply distinguishes a function created by another function from usual functions).

Here's an example of a function that returns closures:

```
incrementer <- function(i) {
  function(x) {
    x + i
  }
}

f <- incrementer(2)
class(f)
```

```
## [1] "function"
```

```
f(1:3)
```

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

```
g <- incrementer(200)
g(1:3)
```

```
## [1] 201 202 203
```

Closure constructors can be powerful tools, but to the uninitiated they can be mysterious. Why is it, for example, that both `f()` and `g()` remember the value `i` was set to when they were created?

The answer is environments. When we see what environment the functions "live" in, we discover it's *not* the global environment. Instead, these functions still use the "temporary" environments created when the function `incrementer()` was invoked; these environments were actually *not* destroyed when the function ended since other functions were created in those environments, then returned, and thus those environments are still needed. We in fact see that the environments the closures use are not the global environment:

```r
environment(incrementer)
```

```
## <environment: R_GlobalEnv>
```

```r
environment(f)
```

```
## <environment: 0x55cdfaa309d8>
```

```r
environment(g)
```

```
## <environment: 0x55cdfab20298>
```

And in fact we see that the `i` values from those function calls live on in those environments.

```r
ls(envir = environment(f))
```

```
## [1] "i"
```

```r
environment(f)$i
```

```
## [1] 2
```

```r
environment(g)$i
```

```
## [1] 200
```

These features give closures their true power. For example, we can create functions that remember how many times they were called.

```r
elephant_func <- function() {  # Cuz elephants never forget
  calls <- 0
  function() {
    calls <<- calls + 1
    calls
  }
}

e1 <- elephant_func()
e1()
```

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

```
e1()
```

```
## [1] 2
```

```
e1()
```

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

```
e2 <- elephant_func()
e2()
```

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

```
e2()
```

```
## [1] 2
```

```
e1()
```

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

This set of functions may be more difficult to unparse than the examples we saw before, yet the only difference in the logic is that the parent environment of, say, `e1()`'s execution environment is not the global environment anymore but instead the environment created upon `e1()`'s birth. This is distinct from the parent environment of any environment created by `e2()`'s execution.

Here's one application of closures. Suppose that we want to examine multiple confidence intervals for any given data set and we want an easy interface for doing so. With closures we can create easy interfaces for changing the parameters of confidence intervals, perhaps changing confidence levels or making them one-sided or two-sided. The following code is a function factory implementing this idea:

```
dat_ci <- function(dat) {
  function(C = 0.95, type = "two.sided") {
    alternative <- switch(type,
      "two.sided" = "two.sided",
      "upper" = "less",
      "lower" = "greater"
    )
    c(t.test(dat, alternative = alternative, conf.level = C)$conf.int)
  }
}

x1 <- rnorm(10)
c1 <- dat_ci(x1)
c1()
```

```
## [1] -1.0975042  0.7559742
```

```
c1(0.9)
```

```
## [1] -0.9217374  0.5802074
```

```
c1(0.99)
```

```
## [1] -1.502127  1.160597
```

```
c1(0.99, "upper")
```

```
## [1]      -Inf 0.985095
```

```
x2 <- rnorm(10)
c2 <- dat_ci(x2)
c2(0.99)
```

```
## [1] -1.312622  1.061951
```

Often in statistics we want to treat a data set as fixed but allow for, say, a parameter, to vary. Closures make doing this easier. Let's consider, for example, the sum of square errors:

$$SSE(\theta) = \sum_{i=1}^{n} (x_i - \theta)^2$$

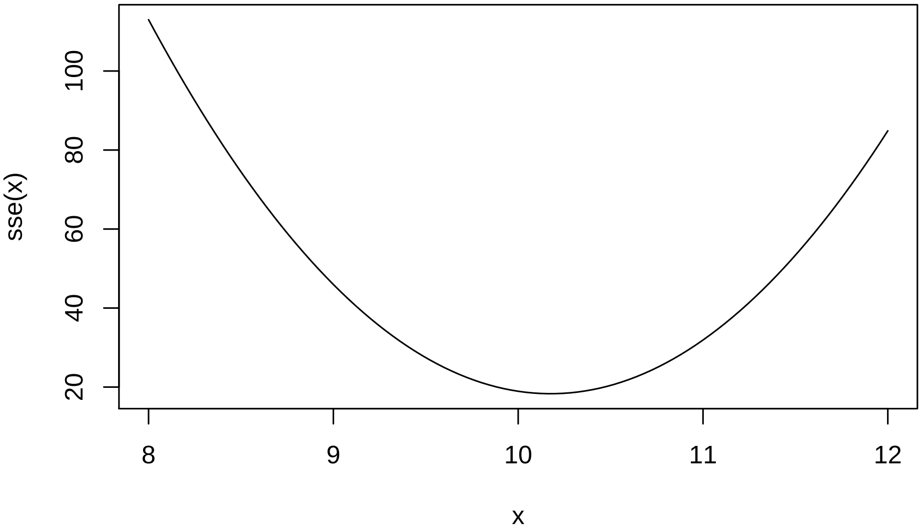
We want to find θ that minimizes the sum of square errors; we might call such a θ a best predictor for an observation, or a good statistic describing the location of x_i. Calculus reveals that the value of θ that minimizes $SSE(\theta)$ is $\theta = \bar{x}$, the sample mean. But let's see if we can avoid calculus.

We can write a function that produces closures that compute $SSE(\theta)$ for input θ given a fixed data set. We would like this function to produce a vector of SSEs if given a vector of values of θ. We do this by actually returning a vectorized version of the $SSE(\theta)$ function using the `Vectorize()` function (which is a functional that returns closures). The final implementation is below:

```
sse_computer <- function(x) {
  f <- function(t) {
    if (!is.numeric(t) || length(t) > 1) stop("Invalid t")
    sum((x - t)^2)
  }
  Vectorize(f)
}
```

If we wished to plot $SSE(\theta)$ for a given data set, we can do so via the `curve()` function, which is a plotting function with an interface friendly to mathematical functions (try `curve(exp, -2, 2)` if you want a quick demo; this is e^x plotted from $x = -10$ to $x = 10$).

```
x <- rnorm(20, mean = 10)
sse <- sse_computer(x)
curve(sse, 8, 12)
```



The curve seems to take a minimum near 10 (which is the known mean of the data.) For reference, the mean of the data is 10.1757534. That said, for what value of θ is $SSE(\theta)$ minimized?

For this we can use the function `optimize()`, which is a numerical optimizer. The first argument to `optimize()` is a function to minimize. (If we wish to maximize a function, recall that maximizing f is the same thing as minimizing $-f$. For this reason, optimization literature generally discusses only minimization problems since such problems automatically include maximization.) The second argument is a vector defining the interval over which the function should be minimized. The function then returns a list with elements `minimum` and `objective`, with `minimum` being where the minimum is attained (in our case, the optimal θ) and `objective` the value of the function at the minimum (here, $SSE(\theta)$).

Optimization here is easy:

```
optimize(sse, c(-100, 100))
```

```
## $minimum
## [1] 10.17575
##
## $objective
## [1] 18.31675
```

```
mean(x)  # For comparison
```

```
## [1] 10.17575
```

These are of course only some uses for closures. I have used them to gain control over random number generation or to write functions that make predictions from fitted models. As the course progresses, look out for more uses of closures (especially after discussing linear models).

Replacement Functions

Consider the following code:

```
vec <- 1:3
names(vec) <- c("a", "b", "c")
vec
```

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

How odd is this code? Doesn't it seem strange that a function call can cause the value of an input to that function to change, simply because of assignment?

Actually, the above code is misleading, for it was not the function `names()` that was called, but `names<-()`.

```
names
```

```
## function (x)  .Primitive("names")
```

```
`names<-`
```

```
## function (x, value)  .Primitive("names<-")
```

A commandment of functional programming is that code should not have side effects. So a function call `f(x)` should not modify the state of variables outside of `f` nor should `x` have its value modified. Following this rule helps make code more easily understood and reasoned about. However, **replacement functions** such as `names<-()` test this principle; while technically adhereing to at least the spirit of the rule, it at least seems as if they change the value of their arguments. (That said, one can still reason easily about the state of the program after this function is called due to the presence of the assignment operator to the right of a function call.)

We write replacement functions like we would any other function, with the following restrictions:

- Our function name should be wrapped in backticks (like with infix functions) and end with `<-`.
- We must have at least two arguments, which we call `x` and `value`, with `x` coming before `value`. They correspond to `f(x) <- value`.
- We may have additional arguments, but any additional arguments must be *between* `x` and `value`.
- The function must return the modified copy of `x`.

I present a simple (yet stupid) example below, where the function changes the values of vectors in particular positions:

```
`modify2<-` <- function(x, value) {
  x[2] <- value
  x
}

x <- 1:10
modify2(x) <- 100
x
```

```
##  [1]   1 100   3   4   5   6   7   8   9  10
```

```
# With a position argument
`modify<-` <- function(x, pos = 1, value) {
  x[pos] <- value
  x
}

modify(x) <- 20
modify(x, 2) <- -2
x
```

```
##  [1] 20 -2  3  4  5  6  7  8  9 10
```

Lecture 3

Object Oriented Programming

Object oriented programming (OOP) is a programming paradigm where the central units in the program are "objects" with associated common data fields (known as **attributes**) and procedures (known as **methods**). Many modern programming languages, including Python, Java, JavaScript, and C++, support OOP. R also supports OOP. However, R does not have a single OOP system, but multiple. R comes with the S3, S4, and RC OOP systems, and packages for other OOP systems (such as **R6**) do exist. Furthermore, OOP in R looks very different from OOP in other programming languages.

R follows functional programming principles first and OOP principles second. Many OOP practices in other language clash with functional programming principles. For examples, methods are often defined with the objects and modify the objects when called; this conflicts with the principle of code having non-obvious side effects. The RC (and **R6**) OOP system most resembles OOP programming in other languages, but veteran R programmers may be shy to use these systems since the produce non-idiomatic code; that is, writing code using these systems produces code that no longer looks like the rest of R, which makes reasoning about the code's behavior more difficult.

The most popular OOP system used in R is S3, followed by S4. S3 is the oldest OOP system used in R, and also the most commonly used. The S3 system was first presented in *Statistical Models in S* (known by the nickname "the white book") by the S language designer J. M. Chambers and coauthor Trevor Hastie. Hadley Wickham, in *Advanced R* said of S3: "S3 is informal and ad hoc, but there is a certain elegance in its minimalism: you can't take away any part of it and still have a useful OO [object oriented] system." S3 is also the only OO system used in the **base** and **stats** packages. Wickham advises to use S3 if no compelling reason to use another system (probably S4) exists.

The S4 system came later, presented in Chambers' book *Programming with Data; A Guide to the S Language* (also known by the nickname "the green book"). S4 keeps the design philosophy of S3 but is less ad hoc and more structured. S4 is a more defensive system and is harder to program with than S3 precisely because it is more structured, and thus less prone to errors and unintended consequences. That said, S4 still resembles S3, so learning S3 is an important step to eventually writing S4 code.

In this lecture I will only discuss the S3 system, since it's both simple and popular. If you continue working with R you're likely to encounter the other OOP systems; S4 comes to mind for me personally. There are plenty of resources for learning those systems, such as *Advanced R* by Hadley Wickham.

Generic Functions

OOP in most programming languages, and in the RC paradigm, generally use method dispatch for developing procedures associated with objects. That is, an object comes with methods the programmer can call to interact with the data in the object. S3, though, handles methods via **generic functions**. A generic function is a function that, when called, can recognize that the object it was called upon is a member of some **class**, which signifies the type of object being worked on. The generic function will then call a function intended for working with objects of that class.

Data frames, for example, are a particular class built upon lists. We can identify the class of a function via the `class()` function.

```r
dat <- data.frame(x = 1:10, y = rnorm(10))
print(dat)
```

```
##     x          y
## 1   1 -0.5535938
## 2   2 -0.2997420
## 3   3  2.6877685
## 4   4 -0.2160176
## 5   5  0.1709505
## 6   6  0.2378153
## 7   7 -1.1541512
## 8   8 -0.5155266
## 9   9 -0.8138550
## 10 10 -1.5297378
```

```r
class(dat)
```

```
## [1] "data.frame"
```

R objects are often accompanied with metadata called **attributes**. These give information about the objects that R functions use. We can look at the attributes of an object with the `attributes()` function, which return all attributes as a list. The `attr()` function accesses a single named attribute. All attributes need to be named to distinguish them from other attributes.

```r
attributes(dat)
```

```
## $names
## [1] "x" "y"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
# More games with attributes: making a matrix from a vector via the dimension
# attribute
(x <- rnorm(10))
```

```
##  [1]  0.65585706  1.13678856 -0.17305632 -0.54889692  0.86632383
##  [6]  0.02828175 -0.62173252 -0.75223397 -1.84525044 -1.31481747
```

```r
attr(x, "dim") <- c(2, 5)
print(x)  # Now behaves like a matrix
```

```
##           [,1]       [,2]       [,3]       [,4]      [,5]
## [1,] 0.6558571 -0.1730563 0.86632383 -0.6217325 -1.845250
## [2,] 1.1367886 -0.5488969 0.02828175 -0.7522340 -1.314817
```

I mention this because whether or not an R object belongs to a class is determined by the `class` attribute, which is merely a string naming the class the object belongs to. In S3 we set an object's class by changing this string. For instance, we can change `dat` to a `list` simply by changing the `class` string to `list`.

```r
attr(dat, "class") <- "list"
print(dat)
```

```
## $x
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $y
##  [1] -0.5535938 -0.2997420  2.6877685 -0.2160176  0.1709505  0.2378153
##  [7] -1.1541512 -0.5155266 -0.8138550 -1.5297378
##
## attr(,"class")
## [1] "list"
## attr(,"row.names")
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Changing the class of an object via `class()` is easier, though.

```r
class(dat) <- "data.frame"
print(dat)
```

```
##     x          y
## 1   1 -0.5535938
## 2   2 -0.2997420
## 3   3  2.6877685
## 4   4 -0.2160176
## 5   5  0.1709505
## 6   6  0.2378153
## 7   7 -1.1541512
## 8   8 -0.5155266
## 9   9 -0.8138550
## 10 10 -1.5297378
```

An implication of this is that we can manually construct a data frame like so:

```
xxx <- list(x = 1:10, y = rnorm(10))
attr(xxx, "row.names") <- c("alpha", "beta", "charlie", "delta", "eagle",
                            "falcon", "gopher", "henry", "iota", "jack")
class(xxx) <- "data.frame"
print(xxx)
```

```
##          x          y
## alpha    1 -0.4605271
## beta     2 -1.8124969
## charlie  3 -0.2506274
## delta    4 -2.1136238
## eagle    5  0.9616646
## falcon   6  0.4150049
## gopher   7  0.1641286
## henry    8 -0.8526285
## iota     9 -0.8526306
## jack    10 -0.2574480
```

Notice, though, that `print()` behaved differently depending on whether `class` was `"list"` or `"data.frame"`. This behavior follows from `print()` being a generic function. This is revealed by looking at the source code for `print()`:

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x55cdf609cee0>
## <environment: namespace:base>
```

The immediate call to the function `UseMethod()` declares `print()` as a generic function. See, methods in R are not stored with data like in other languages (and RC/R6) but with functions. We can see all the methods associated with a generic function via the `methods()` function.

```
methods(print)
```

```
##   [1] print.abbrev*
##   [2] print.acf*
##   [3] print.AES*
##   [4] print.anova*
##   [5] print.Anova*
##   [6] print.anova.loglm*
##   [7] print.aov*
##   [8] print.aovlist*
##   [9] print.ar*
##  [10] print.Arima*
##  [11] print.arima0*
##  [12] print.AsIs
##  [13] print.aspell*
##  [14] print.aspell_inspect_context*
##  [15] print.bibentry*
##  [16] print.Bibtex*
##  [17] print.browseVignettes*
##  [18] print.by
##  [19] print.bytes*
##  [20] print.changedFiles*
##  [21] print.check_code_usage_in_package*
##  [22] print.check_compiled_code*
##  [23] print.check_demo_index*
##  [24] print.check_depdef*
##  [25] print.check_details*
##  [26] print.check_details_changes*
##  [27] print.check_doi_db*
##  [28] print.check_dotInternal*
##  [29] print.check_make_vars*
##  [30] print.check_nonAPI_calls*
##  [31] print.check_package_code_assign_to_globalenv*
##  [32] print.check_package_code_attach*
##  [33] print.check_package_code_data_into_globalenv*
##  [34] print.check_package_code_startup_functions*
##  [35] print.check_package_code_syntax*
##  [36] print.check_package_code_unload_functions*
##  [37] print.check_package_compact_datasets*
##  [38] print.check_package_CRAN_incoming*
##  [39] print.check_package_datasets*
##  [40] print.check_package_depends*
##  [41] print.check_package_description*
##  [42] print.check_package_description_encoding*
##  [43] print.check_package_license*
##  [44] print.check_packages_in_dir*
##  [45] print.check_packages_used*
##  [46] print.check_po_files*
##  [47] print.check_pragmas*
##  [48] print.check_Rd_contents*
##  [49] print.check_Rd_line_widths*
##  [50] print.check_Rd_metadata*
##  [51] print.check_Rd_xrefs*
##  [52] print.check_RegSym_calls*
##  [53] print.check_S3_methods_needing_delayed_registration*
##  [54] print.check_so_symbols*
##  [55] print.check_T_and_F*
##  [56] print.check_url_db*
##  [57] print.check_vignette_index*
##  [58] print.checkDocFiles*
##  [59] print.checkDocStyle*
##  [60] print.checkFF*
##  [61] print.checkRd*
##  [62] print.checkReplaceFuns*
##  [63] print.checkS3methods*
##  [64] print.checkTnF*
##  [65] print.checkVignettes*
##  [66] print.citation*
##  [67] print.codoc*
##  [68] print.codocClasses*
##  [69] print.codocData*
##  [70] print.colorConverter*
##  [71] print.compactPDF*
##  [72] print.condition
##  [73] print.connection
##  [74] print.correspondence*
##  [75] print.CRAN_package_reverse_dependencies_and_views*
##  [76] print.data.frame
##  [77] print.Date
##  [78] print.default
##  [79] print.dendrogram*
##  [80] print.density*
##  [81] print.difftime
##  [82] print.dist*
##  [83] print.Dlist
##  [84] print.DLLInfo
##  [85] print.DLLInfoList
##  [86] print.DLLRegisteredRoutines
##  [87] print.document_context*
##  [88] print.document_position*
##  [89] print.document_range*
##  [90] print.document_selection*
##  [91] print.dummy_coef*
##  [92] print.dummy_coef_list*
##  [93] print.ecdf*
##  [94] print.eigen
##  [95] print.factanal*
##  [96] print.factor
##  [97] print.family*
##  [98] print.fileSnapshot*
##  [99] print.findLineNumResult*
## [100] print.fitdistr*
## [101] print.formula*
## [102] print.fractions*
## [103] print.frame*
## [104] print.fseq*
## [105] print.ftable*
## [106] print.function
## [107] print.gamma.shape*
## [108] print.getAnywhere*
## [109] print.glm*
## [110] print.glm.dose*
## [111] print.hclust*
## [112] print.help_files_with_topic*
## [113] print.hexmode
## [114] print.HoltWinters*
## [115] print.hsearch*
## [116] print.hsearch_db*
## [117] print.htest*
## [118] print.html*
## [119] print.html_dependency*
## [120] print.infl*
## [121] print.integrate*
## [122] print.isoreg*
## [123] print.kmeans*
## [124] print.knitr_kable*
## [125] print.Latex*
## [126] print.LaTeX*
## [127] print.lda*
## [128] print.libraryIQR
## [129] print.listof
## [130] print.lm*
## [131] print.loadings*
## [132] print.loess*
## [133] print.logLik*
## [134] print.loglm*
## [135] print.lqs*
## [136] print.ls_str*
## [137] print.mca*
## [138] print.medpolish*
## [139] print.MethodsFunction*
## [140] print.mtable*
## [141] print.NativeRoutineList
## [142] print.news_db*
## [143] print.nls*
## [144] print.noquote
## [145] print.numeric_version
## [146] print.object_size*
## [147] print.octmode
## [148] print.packageDescription*
## [149] print.packageInfo
## [150] print.packageIQR*
## [151] print.packageStatus*
## [152] print.pairwise.htest*
## [153] print.person*
## [154] print.polr*
## [155] print.POSIXct
## [156] print.POSIXlt
## [157] print.power.htest*
## [158] print.ppr*
## [159] print.prcomp*
## [160] print.princomp*
## [161] print.proc_time
## [162] print.qda*
## [163] print.quosure*
## [164] print.quosures*
## [165] print.raster*
## [166] print.Rcpp_stack_trace*
## [167] print.Rd*
## [168] print.recordedplot*
## [169] print.restart
## [170] print.RGBcolorConverter*
## [171] print.ridgelm*
## [172] print.rlang_box_done*
## [173] print.rlang_box_splice*
## [174] print.rlang_data_pronoun*
## [175] print.rlang_envs*
## [176] print.rlang_error*
## [177] print.rlang_fake_data_pronoun*
## [178] print.rlang_lambda_function*
## [179] print.rlang_trace*
## [180] print.rlang_zap*
## [181] print.rle
## [182] print.rlm*
## [183] print.rms.curv*
## [184] print.roman*
## [185] print.sessionInfo*
## [186] print.shiny.tag*
## [187] print.shiny.tag.list*
## [188] print.simple.list
## [189] print.smooth.spline*
## [190] print.socket*
## [191] print.srcfile
## [192] print.srcref
## [193] print.stepfun*
## [194] print.stl*
## [195] print.StructTS*
## [196] print.subdir_tests*
## [197] print.summarize_CRAN_check_status*
## [198] print.summary.aov*
## [199] print.summary.aovlist*
## [200] print.summary.ecdf*
## [201] print.summary.glm*
## [202] print.summary.lm*
## [203] print.summary.loess*
## [204] print.summary.loglm*
## [205] print.summary.manova*
## [206] print.summary.negbin*
## [207] print.summary.nls*
## [208] print.summary.packageStatus*
## [209] print.summary.polr*
## [210] print.summary.ppr*
## [211] print.summary.prcomp*
## [212] print.summary.princomp*
## [213] print.summary.rlm*
## [214] print.summary.table
## [215] print.summary.warnings
## [216] print.summaryDefault
## [217] print.table
## [218] print.tables_aov*
## [219] print.terms*
## [220] print.ts*
## [221] print.tskernel*
## [222] print.TukeyHSD*
## [223] print.tukeyline*
## [224] print.tukeysmooth*
## [225] print.undoc*
## [226] print.vignette*
## [227] print.warnings
## [228] print.xfun_raw_string*
## [229] print.xfun_strict_list*
## [230] print.xgettext*
## [231] print.xngettext*
## [232] print.xtabs*
## see '?methods' for accessing help and source code
```

Suppose `print()` is called on an object `x` with class `"foo"`. The actual function `print()` is a generic function, so it won't do any printing. Instead, it will look for a function called `print.foo()`. If it finds such a function, that function will be called. Otherwise, `print.default()` will be called. Thus, when `print()` was called on `dat` when `dat` had class `"data.frame"`, `print()` called the function `print.data.frame()` on `dat`.

```
print.data.frame
```

```
## function (x, ..., digits = NULL, quote = FALSE, right = TRUE,
##     row.names = TRUE, max = NULL)
## {
##     n <- length(row.names(x))
##     if (length(x) == 0L) {
##         cat(sprintf(ngettext(n, "data frame with 0 columns and %d row",
##             "data frame with 0 columns and %d rows"), n), "\n",
##             sep = "")
##     }
##     else if (n == 0L) {
##         print.default(names(x), quote = FALSE)
##         cat(gettext("<0 rows> (or 0-length row.names)\n"))
##     }
##     else {
##         if (is.null(max))
##             max <- getOption("max.print", 99999L)
##         if (!is.finite(max))
##             stop("invalid 'max' / getOption(\"max.print\"): ",
##                 max)
##         omit <- (n0 <- max%/%length(x)) < n
##         m <- as.matrix(format.data.frame(if (omit)
##             x[seq_len(n0), , drop = FALSE]
##         else x, digits = digits, na.encode = FALSE))
##         if (!isTRUE(row.names))
##             dimnames(m)[[1L]] <- if (isFALSE(row.names))
##                 rep.int("", if (omit)
##                   n0
##                 else n)
##             else row.names
##         print(m, ..., quote = quote, right = right, max = max)
##         if (omit)
##              cat(" [ reached 'max' / getOption(\"max.print\") -- omitted",
##                  n - n0, "rows ]\n")
##      }
##      invisible(x)
## }
## <bytecode: 0x55cdfaac7530>
## <environment: namespace:base>
```

This is the reason why R objects' actual structure as a `list` or `vector` gets obfuscated when the object is printed; the object has a `class` attribute for which a `print()` method exists, causing that method to be called and the function to appear to behave differently.

For example, functions such as `t.test()` and `prop.test()` return `list`s of class `"htest"`, and since there is a `print()` method called `print.htest()`, that function is called whenever these objects are printed.

```
x <- rnorm(10)
res <- t.test(x)
print(res)
```

```
##
##  One Sample t-test
##
## data:  x
## t = 1.8984, df = 9, p-value = 0.09012
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -0.07536256  0.86194657
## sample estimates:
## mean of x
##  0.393292
```

```
class(res)
```

```
## [1] "htest"
```

```
class(res) <- NULL  # Deleting the class attribute; now res is just a list
print(res)
```

```
## $statistic
##        t
## 1.898388
##
## $parameter
## df
##  9
##
## $p.value
## [1] 0.09012002
##
## $conf.int
## [1] -0.07536256  0.86194657
## attr(,"conf.level")
## [1] 0.95
##
## $estimate
## mean of x
##  0.393292
##
## $null.value
## mean
##    0
##
## $stderr
## [1] 0.2071715
##
## $alternative
## [1] "two.sided"
##
## $method
## [1] "One Sample t-test"
##
## $data.name
## [1] "x"
```

S3 objects can be members of multiple classes; in such cases, the class attribute is a character vector with length greater than one. The order of the strings in the vector matter, as generic functions will look through the vector starting from first and ending at last when looking for a method to call. The moment a matching class is found, all other class memberships are ignored. This behavior allows S3 to support the OOP notion of **inheritence**, where class A **inherits** from class B if every member of class A is also a member of class B. This means that any methods that work for class B also work for class A.

For instance, consider `data.table` objects from the **data.table** library (which you should look into; it's my preferred library for "replacing" R's bare bones data frames). This library produces `data.frame`-like data structures, but with an interface that's easier to use.

```r
library(data.table)
dtab <- data.table(x = 1:1000, y = rnorm(1000))
print(dtab)
```

```
##          x          y
##    1:    1  0.4825888
##    2:    2  1.6628911
##    3:    3  1.8131992
##    4:    4  0.6434752
##    5:    5  0.7516509
##   ---
##  996:  996  1.8417046
##  997:  997 -1.0216248
##  998:  998  0.6574799
##  999:  999  0.2680119
## 1000: 1000  1.4658343
```

```
class(dtab)
```

```
## [1] "data.table" "data.frame"
```

Above, `dtab` is both a `"data.table"` and a `"data.frame"`, but if we reverse the ordering of the classes in `class` attribute vector, we get different `print()` behavior.

```
print(dtab[1:10,])
```

```
##      x          y
##  1:  1  0.4825888
##  2:  2  1.6628911
##  3:  3  1.8131992
##  4:  4  0.6434752
##  5:  5  0.7516509
##  6:  6  0.6794129
##  7:  7  3.0876051
##  8:  8 -0.1324895
##  9:  9  1.3270360
## 10: 10 -0.5232304
```

```
class(dtab) <- rev(class(dtab))
print(dtab[1:10,])
```

```
##     x          y
## 1   1  0.4825888
## 2   2  1.6628911
## 3   3  1.8131992
## 4   4  0.6434752
## 5   5  0.7516509
## 6   6  0.6794129
## 7   7  3.0876051
## 8   8 -0.1324895
## 9   9  1.3270360
## 10 10 -0.5232304
```

```
class(dtab) <- rev(class(dtab))  # Restoring dtab to its original state
```

Many of the functions you use are generic functions. These functions include `print()`, `summary()`, `plot()`, `mean()`, and many more. Extending a generic function for a new class (or even just changing its behavior for an existing class) is also easy. If `foo()` is a generic function and we want a method for objects of class `bar`, we merely need to write the function `foo.bar()`. (This is why we discourage using `.` in names; we want to keep that character only in the names of function methods. Otherwise, we can't tell if `print.data.frame()` is a method of `print()` for objects of class `data.frame`, or if it's a method of the non-existant generic function `print.data()` for objects of class `frame`, or if it's just a stand-alone function `print.data.frame()`.) We can call `foo.bar()` without ever calling `foo()`, but this is discouraged since these functions never check that their input object is actually of the intended class.

This, by the way, marks one problem with the S3 OO system: object assumptions are never defined and almost never checked. Changing an object's class is as simple as changing the `class` string. Thus we can get unintended consequences, such as with this malicious code:

```
# Despite not being produced by the appropriate function, this object gets
# assigned to class htest anyway
not_htest <- list(x = "a", y = 2)
print(not_htest)
```

```
## $x
## [1] "a"
##
## $y
## [1] 2
```

```
class(not_htest) <- "htest"
print(not_htest)  # This time print.htest() was called, and fails miserably
```

```
##
##
##
## data:
```

Programmers have to trust users to not abuse their OO systems, and simple trust is never a good thing in programming.

Suppose we want a new generic function. For example, let's create a generic function `collapse()`, which attempts to "collapse" objects into a length-one vector. We first declare that the function `collapse()` is generic like so:

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

The responsible thing to do is define a `default` method that will be called when objects without a class for which a `collapse()` method exists are passed to `collapse()`.

```
collapse.default <- function(x) {x[[1]]}
```

Now we can define some methods for specific classes of objects. These methods are what make the `collapse()` function "useful".

```
collapse.numeric <- function(x) {sum(x)}
collapse.integer <- function(x) {collapse.numeric(x)}
collapse.logical <- function(x) {collapse.numeric(x)}
collapse.character <- function(x) {paste0(x, collapse = "")}
collapse.list <- function(x) {collapse(c(unlist(x)))}
collapse.data.frame <- function(x) {collapse.list(x)}
```

Notice the resulting behavior:

```
collapse(1:10)
```

```
## [1] 55
```

```
collapse(1:10 <= 5)
```

```
## [1] 5
```

```
collapse(c("a", "b", "c"))
```

```
## [1] "abc"
```

```
collapse(res)   # A plain list
```

```
## [1] "1.8983882539705890.0901200216705836-0.07536256226954690.8619465694774080.393292003603931
```

```
collapse(dat)   # Our data frame from before
```

```
## [1] 53.01391
```

```
collapse(dtab)  # A data.table but also a data.frame
```

```
## [1] 500504.1
```

```
collapse(iris$Species)  # A factor object; no method exists for factors
```

```
## [1] setosa
## Levels: setosa versicolor virginica
```

(Actually the above code swept something under the rug: many of the objects created above did not have a `class` attribute. What they did have was a base type, revealed with the function `typeof()`, which is what the `class()` function actually returns. These types are built into R and predate even S3, since they are fundamentally how R works. We can treat these as if they are a class anyway, though.)

Creating a Class

Creating a class is as simple as creating some object (often a `list`, the most flexible R data structure) and giving it a `class` attribute. We then could throw on some methods for existing generic functions. I could leave the discussion at that, but let's instead see a worked example for creating a new class. We will also see some advanced techniques like operator overloading.

In this example, we will be creating a class to emulate an account, like a banking account. In our program we have a simple model for what it means to be an account: an account is an ordered list of transactions, starting with an initial transaction that is the initial amount of money the account starts with. After defining an account, we should define a transaction. Here, we will define a transaction, at minimum, to consist of a date and an amount. Amounts below zero represent withdrawals, while amounts above zero represent deposits. However, we may want to augment our transactions with a "memo" field, which could list a transaction counterparty or simply be a reminder for why the transaction exists. Similarly, we should augment our account with a name attribute and an owner attribute to help distinguish different accounts, such as checking and savings, and to allow for the concept of people being associated with accounts.

Thus our accounting OO system should include two classes of objects: accounts and transactions. Since accounts are built from transactions, we will first work on transaction objects. We start by creating a function that produces transactions objects, which we call a **constructor** function.

```r
transaction <- function(date, amount, memo = "") {
  obj <- list(date = as.Date(date),
              amount = as.numeric(amount),
              memo = as.character(memo)
  )
  class(obj) <- "transaction"
  obj
}
```

Recall functions such as `is.numeric()` or `is.data.frame()`? I consider writing such a function for a new class a good programming practice. So let's define such a function here.

```r
is.transaction <- function(x) {class(x) == "transaction"}
```

Let's create a transaction now, and see what happens:

```r
a <- transaction("2010-01-01", 10, memo = "Hello, world!")
is.transaction(a)
```

```
## [1] TRUE
```

```
a
```

```
## $date
## [1] "2010-01-01"
##
## $amount
## [1] 10
##
## $memo
## [1] "Hello, world!"
##
## attr(,"class")
## [1] "transaction"
```

Let's go ahead and write a `print()` method for our object, for pretty printing.

```
print.transaction <- function(x, space = 10) {
  tdate <- as.character(x$date)
  datestring <- paste0("    ", tdate, ":")
  formatstring <- paste0("%+", space[[1]], ".2f")  # See sprintf() to explain
  amountstring <- sprintf(formatstring, x$amount)
  if (x$memo == "") {
    memostring <- ""
  } else {
    memostring <- paste0("(", x$memo, ")")
  }
  cat(datestring, amountstring, memostring, "\n")
}
```

Then when we print the transaction we get a much nicer output.

```
a
```

```
##     2010-01-01:     +10.00 (Hello, world!)
```

Users may want to access `date`, `amount`, and `memo` fields and modify them easily; we will create helper functions and allow for easy modification of them.

```
transaction_date <- function(trns) {
  trns$date
}

`transaction_date<-` <- function(trns, value) {
  trns$date <- as.Date(value)
  trns
}

amount <- function(trns) {
  trns$amount
```

```r
}

`amount<-` <- function(trns, value) {
  trns$amount <- as.numeric(value)
  trns
}

memo <- function(trns) {
  trns$memo
}

`memo<-` <- function(trns, value) {
  trns$memo <- as.character(value)
  trns
}
```

Let's see these functions in action:

```r
transaction_date(a)
```

```
## [1] "2010-01-01"
```

```r
amount(a)
```

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

```r
memo(a)
```

```
## [1] "Hello, world!"
```

```r
transaction_date(a) <- "2020-01-01"
amount(a) <- -20
memo(a) <- "Vacation money"

a
```

```
##     2020-01-01:     -20.00 (Vacation money)
```

Okay, the transaction object looks good so far. Let's next start creating an account object. This again will be built off of a list. We will start again with a constructor function and an `is`-type function.

```r
account <- function(start, owner, init = 0, title = "Account") {
  obj <- list(
    title = as.character(title),
    owner = as.character(owner),
    transactions = list(transaction(start, init, "Initial"))
  )
  class(obj) <- "account"
  obj
}

is.account <- function(x) {class(x) == "account"}

acc <- account("2010-01-01", "John Doe")
acc
```

```
## $title
## [1] "Account"
##
## $owner
## [1] "John Doe"
##
## $transactions
## $transactions[[1]]
##     2010-01-01:      +0.00 (Initial)
##
##
## attr(,"class")
## [1] "account"
```

```
is.account(acc)
```

```
## [1] TRUE
```

Before carrying on, we should think about the list of transactions more. This list should be sorted so that the initial amount is always the first element of the list and the remaining elements are sorted in order of date. We also should not have transactions before the initial date. We should write a sorting function that, given the list of transactions, will put them in the proper order, and fail if there is any transaction with a date prior to the initial date.

The function `sort()` is a generic function, so we could use it for sorting the transaction list. Before we do that, though, let's write functions that pull important information from accounts. We'll start with account title and owner.

```
account_title <- function(account) {
  account$title
}

`account_title<-` <- function(account, value) {
  account$title <- as.character(value)
  account
}

account_owner <- function(account) {
  account$owner
}
```

```
`account_owner<-` <- function(account, value) {
  account$owner <- as.character(value)
  account
}

account_transactions <- function(account) {
  account$transactions
}

`account_transactions<-` <- function(account, value) {
  account$transactions <- value
  account
}
```

We'll additionally create functions that check that all objects in the `transactions` list are `transaction`-class objects, and functions that pull all transaction dates, amounts, and memos.

```
all_transactions <- function(account) {
  all(sapply(account_transactions(account), is.transaction))
}

account_dates <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  Reduce(c, lapply(account_transactions(account), transaction_date))
}

account_trans_amounts <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  sapply(account_transactions(account), amount)
}

account_memos <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  sapply(account_transactions(account), memo)
}

transaction_count <- function(account) {
  length(account_transactions(account))
}
```

Many of these functions seem simple, to the point that we may question why they exist. In fact, it seems that if our objective is to save time typing, we should not use these functions (or at least use shorter names). But there's good reason to have functions like these. First, these functions are abstractions. If we decide to change how `account` objects store their data, many of these functions will still work so long as earlier functions are changed to account for the knew design. Thus, we've made our program more flexible. Second, this should be easier for programmers to read. They only need to know that the function `account_transactions()` gets a list of transactions from the account, without knowing how exactly that works; also, they don't know how `all_transactions()` works, but they can reason that the function checks that everything in a list is a `transaction`-class object. Third, by providing users with these interface functions, we've given users the tools they need to write safe code that doesn't accidentally break our object. The user can be reassured that `account_title()` and `account_owner()` will handle the `title` and `owner` attributes of the object properly. Additionally, the user can be reassured that `account_title()` and `account_owner()` will always work the same way in future versions of the code, while `account$title` or `account$owner` depend on the specific structure of the object and thus are not safe to use, since they could change in the future. This allows the user to write robust code that's less likely to break due to unforseen changes. Additionally, we as developers of the object gain license to change the specific structure of the object so long as these functions work the same for all future versions of the code.

Now that we have these helper functions, we can write a `sort()` method.

```r
sort.account <- function(x, decreasing = FALSE, ...) {
  # There might be multiple entries with memo Initial; design code for that
  memo_Initial <- which(account_memos(x) == "Initial")
  if (length(memo_Initial) == 0) {
    date_Initial <- min(account_dates(x))
    true_Initial <- which.min(account_dates(x))
  } else {
    date_Initial <- account_dates(x)[memo_Initial]
    true_Initial <- which((account_dates(x) == min(date_Initial)) &
                          (account_memos(x) == "Initial"))
  }
  tcount <- transaction_count(x)
  nix <- (1:tcount)[-true_Initial]
  ordered_nix <- nix[order(account_dates(x)[nix], decreasing = decreasing, ...)]
  if (decreasing) {
    final_order <- c(ordered_nix, true_Initial)
  } else {
    final_order <- c(true_Initial, ordered_nix)
  }
  account_transactions(x) <- account_transactions(x)[final_order]
  x
}
```

If there is no transaction with memo line `"Initial"`, or if there is a transaction before the `"Initial"` transaction, we consider the object malformed, and we want to detect that. Let's write code detecting this.

```
bad_Initial <- function(account) {
  if (!all_transactions(account)) stop("Not all transactions of right class")
  if (!("Initial" %in% account_memos(account))) stop("No Initial transaction")

  memo_Initial <- which(account_memos(account) == "Initial")
  date_Initial <- min(account_dates(account)[memo_Initial])
  sort(account_dates(account))[1] < date_Initial
}
```

Currently the `print()` method produces ugly output; let's write a better method:

```
print.account <- function(x, presort = FALSE) {
  if (presort) {
    x <- sort(x)
  }
  cat("Title:", account_title(x))
  cat("\nOwner:", account_owner(x))
  cat("\nTransactions:\n----------------------------------------------------\n")
  for (t in account_transactions(x)) {
    print(t)
  }
}
acc
```

```
## Title: Account
## Owner: John Doe
## Transactions:
## ----------------------------------------------------
##     2010-01-01:      +0.00 (Initial)
```

We would also like a `summary()` method. The purpose of `print()` is to give us the information stored in the object. With `summary()` we would like basic information such as how much money is currently in the account, the number of transactions, the account origination date, recent transactions, and perhaps more. What we *should* do, though, is return a list with a special class, such as `summary.account`, so that a `print()` method is responsible for presenting summary information to the user while the `summary()` method is responsible for collecting it. This allows the user to save the information `summary()` obtains.

```
summary.account <- function(object, recent = 5) {
  if (bad_Initial(object)) warning("account object malformed!")
  res <- list()
  res$title <- account_title(object)
  res$owner <- account_owner(object)
  res$balance <- sum(account_trans_amounts(object))
  res$tcount <- transaction_count(object)
  res$rtrans <- account_transactions(sort(object,
                                          decreasing = TRUE)
                                    )[1:min(recent, res$tcount)]

  class(res) <- "summary.account"
  res
}

print.summary.account <- function(x, prefix = "\t") {
  cat("\n")
  cat(strwrap(x$title, prefix = prefix), sep = "\n")
  cat("\n")
  cat("Owner:  ", sprintf("%20s", x$owner), "\n", sep = "")
  cat("Transactions:  ", sprintf("%13s", x$tcount), "\n", sep = "")
  cat("Balance:  ", sprintf("%18.2f", x$balance), "\n", sep = "")
  cat("\nRecent Transactions:\n----------------------------------------------------\n")
  for (t in x$rtrans) {
    print(t)
  }
}

summary(acc)
```

```
##
##  Account
##
## Owner:              John Doe
## Transactions:             1
## Balance:               0.00
##
## Recent Transactions:
## ----------------------------------------------------
##     2010-01-01:      +0.00 (Initial)
```

We could write methods for many more functions, such as `plot()`, but what we need now is a way to add transactions to the account. For this, we will overload the `+` operator. **Operator overloading** is the practice of taking some operator, such as `+` or `*` or `&`, and extending its meaning to a context for which a use was not originally forseen. We do this by treating `+` as a generic function.

```
`+.account` <- function(x, y) {
  account_transactions(x) <- c(account_transactions(x), account_transactions(y))
  sort(x)
}
```

One unfortunate feature of this code is that `+` is not quite commutative; `x + y` is different from `y + x`. While the transactions are the same, the title and owner of the account depends on order. We would need more intelligent code to fix this.

Ideally we would like to add transactions to accounts, but there is no way to currently do that. We should create a function `as.account()` that can easily turn transactions into accounts. We should also make the function generic; we might think of other objects in the future we could coerce into accounts.

```
as.account <- function(x, ...) UseMethod("as.account")
as.account.transaction <- function(x, title = "Transaction", owner = "noone") {
  res <- account(start = transaction_date(x), init = amount(x), owner = owner,
                 title = title)
  memo(account_transactions(res)[[1]]) <- memo(x)
  res
}
```

Now if we want to add transactions to the account, we can do so like so:

```
acc + as.account(a)
```

```
## Title: Account
## Owner: John Doe
## Transactions:
## ----------------------------------------------------
##     2010-01-01:      +0.00 (Initial)
##     2020-01-01:     -20.00 (Vacation money)
```

Let’s now add some fictitious transactions to the account. We will store them in a data frame then add them to the account.

```
dat <- data.frame("date" = seq(as.Date("2010-01-02"), as.Date("2010-01-31"),
                               by = "day"),
                  "amount" = rnorm(31 - 1),
                  "memo" = paste("Transaction", 1:(31 - 1)))
acc <- acc + Reduce(`+`, lapply(1:nrow(dat), function(i) {
         r <- dat[i, ]
         as.account(transaction(date = r$date, amount = r$amount,
                                memo = r$memo))
       }))
acc
```

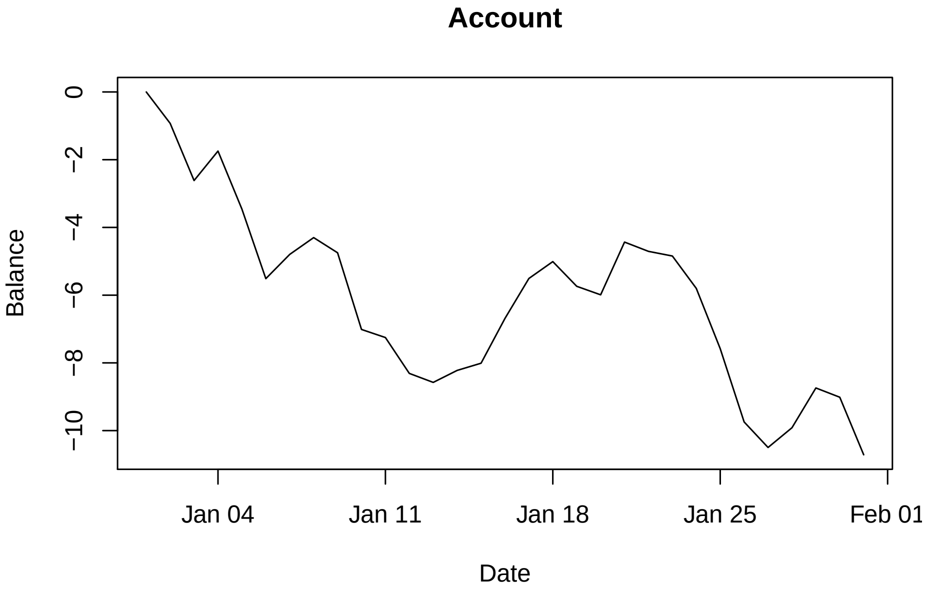
```
## Title: Account
## Owner: John Doe
## Transactions:
## ----------------------------------------------------
##     2010-01-01:      +0.00 (Initial)
##     2010-01-02:      -0.93 (Transaction 1)
##     2010-01-03:      -1.69 (Transaction 2)
##     2010-01-04:      +0.87 (Transaction 3)
##     2010-01-05:      -1.71 (Transaction 4)
##     2010-01-06:      -2.05 (Transaction 5)
##     2010-01-07:      +0.72 (Transaction 6)
##     2010-01-08:      +0.50 (Transaction 7)
##     2010-01-09:      -0.45 (Transaction 8)
##     2010-01-10:      -2.26 (Transaction 9)
##     2010-01-11:      -0.24 (Transaction 10)
##     2010-01-12:      -1.06 (Transaction 11)
##     2010-01-13:      -0.27 (Transaction 12)
##     2010-01-14:      +0.35 (Transaction 13)
##     2010-01-15:      +0.21 (Transaction 14)
##     2010-01-16:      +1.32 (Transaction 15)
##     2010-01-17:      +1.18 (Transaction 16)
##     2010-01-18:      +0.50 (Transaction 17)
##     2010-01-19:      -0.73 (Transaction 18)
##     2010-01-20:      -0.25 (Transaction 19)
##     2010-01-21:      +1.56 (Transaction 20)
##     2010-01-22:      -0.27 (Transaction 21)
##     2010-01-23:      -0.14 (Transaction 22)
##     2010-01-24:      -0.96 (Transaction 23)
##     2010-01-25:      -1.78 (Transaction 24)
##     2010-01-26:      -2.16 (Transaction 25)
##     2010-01-27:      -0.75 (Transaction 26)
##     2010-01-28:      +0.58 (Transaction 27)
##     2010-01-29:      +1.17 (Transaction 28)
##     2010-01-30:      -0.27 (Transaction 29)
##     2010-01-31:      -1.70 (Transaction 30)
```

```r
summary(acc)
```

```
##
##  Account
##
## Owner:               John Doe
## Transactions:              31
## Balance:               -10.71
##
## Recent Transactions:
## -----------------------------------------------------
##     2010-01-31:      -1.70 (Transaction 30)
##     2010-01-30:      -0.27 (Transaction 29)
##     2010-01-29:      +1.17 (Transaction 28)
##     2010-01-28:      +0.58 (Transaction 27)
##     2010-01-27:      -0.75 (Transaction 26)
```

Let's end by creating a `plot()` method for accounts.

```
plot.account <- function(x, y, ...) {
  if (bad_Initial(x)) stop("Malformed account object")
  x <- sort(x)
  unique_dates <- unique(account_dates(x))
  date_trans_sum <- sapply(unique_dates, function(d) {
    idx <- which(account_dates(x) == d)
    sum(account_trans_amounts(x)[idx])
  })
  plot(unique_dates, cumsum(date_trans_sum), type = "l",
       main = account_title(x), xlab = "Date", ylab = "Balance", ...)
}

plot(acc)
```



And there's so much more we could start doing with this construct, such as writing functions to read transactions and form accounts from CSV files, extend the generic function `as.account()`, add an `as.transaction()` generic functions and give it methods, etc. We should end the lecture here, though, as we now have a sensible and functioning class system, which is what we wanted.

Lecture 4

One-Way Analysis of Variance (ANOVA)

Procedures such as the two-sample t-test exist to compare the means of two distinct populations. But what if we want to compare the means of more than just two populations? When doing so, we often wish to discern:

- whether there is a difference in the means of any of the populations' means; and
- if there is a difference, which means are different.

A naïve first attempt would perform two-sample t-tests for every combination of two populations. If there are K populations, there would be $\binom{K}{2}$ tests. Aside for there being many separate tests (and thus a lot of work), when one conducts many tests like this, the chances of making a Type I error on any test are high, often much higher than the specified Type I error rate α. The **Bonferroni correction** would suggest we divide the significance level by the number of tests done, thus rejecting one of the null hypotheses if the p-value drops below $\alpha/\binom{K}{2}$. This correction, however, is quite drastic, perhaps too conservative.

Another solution, at least for determining whether any of the means are different, is to perform what's known as a "overall" test. In this case, it would determine whether any of the means are different from each other. Depending on the result of the overall test, we would then look to determine which means differ. (If the test does not reject the null hypothesis of no difference we would not proceed with a detailed analysis.)

Analysis of variance (ANOVA) is a statistical procedure looking to address the first issue: whether there is a difference in means among the populations. Later we will look at the other issue.

Suppose there are K populations; thus, there are K means, $\mu_1, \mu_2, \cdots, \mu_K$. ANOVA seeks to decide between:

$$H_0 : \mu_1 = \cdots = \mu_K = \mu$$

$$H_A: \text{ there exists } i,j \text{ s.t. } \mu_i \neq \mu_j$$

However, the ANOVA procedure is seen as doing more than just deciding between two hypotheses. In fact, we're estimating the statistical model:

$$x_{ik} = \mu_k + \epsilon_{ik}$$

where $i \in \{1, \ldots, n_k\}$ and $n_1 + \cdots + n_K = N$. The model listed above is the **one-way ANOVA** model, since the populations differ in only one aspect.

ANOVA assumes that for all i and k, $\epsilon_{ik} \sim N(0, \sigma^2)$. We call the terms ϵ_{ik} the **residuals** of the model. The normality of the residuals matters for smaller sample sizes, but less so for larger sample sizes. But the assumption of common variance matters a great deal, regardless of sample size. Thus we must always check it. Statisticians often use box plots to judge whether the common variance assumption is appropriate.

Let $\bar{x}_k = \frac{1}{n_k}\sum_{i=1}^{n_k} x_{ik}$ and $\bar{x}. = \frac{1}{N}\sum_{k=1}^{K}\sum_{i=1}^{n_k} x_{ik}$ $SSE = \sum_{k=1}^{K}\sum_{i=1}^{n_k}(x_{ik} - \bar{x}_k)^2$ and $SSTr = \sum_{k=1}^{K}(\bar{x}_k - \bar{x}.)^2$. Let $\nu_n = K - 1$ and $\nu_d = N - K$ be the numerator and denominator degrees of freedom, respectively. Then the ANOVA test statistic is:

$$f = \frac{SSE/\nu_n}{SSTr/\nu_d}$$

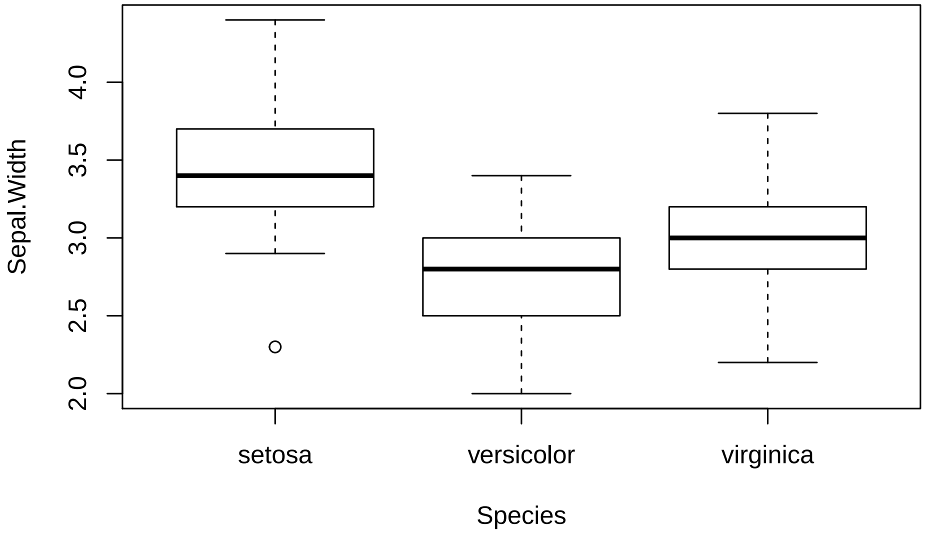
The distribution of f if H_0 is true is the F-distribution F_{ν_n,ν_d} distribution, the F distribution with numerator degrees of freedom ν_n and denominator degrees of freedom ν_d. The p-value is $P(F_{\nu_n,\nu_d} > f)$.

A number of R functions can perform ANOVA, particularly `oneway.test()`, `aov()`, and `lm()`.

`oneway.test()`

Consider the `iris` data set. Due to the assumption that the populations have a common variance, we should check with a boxplot whether the assumption seems plausible.

```
boxplot(Sepal.Width ~ Species, data = iris)
```



The spread of the data sets are similar; furthermore, the boxplot does suggest that there could be a difference in means. We instruct `oneway.test()` to perform ANOVA via a command resembling `oneway.test(x ~ f, data = d)`, where `x` is the variable we test, `f` identifies the populations, and `d` is a data frame containing the variables `x` and `f` (in long-form format). Note that `x` *must* be numeric and `f` *must* be a factor variable. This holds throughout the lecture.

```
oneway.test(Sepal.Length ~ Species, data = iris)
```

```
##
##  One-way analysis of means (not assuming equal variances)
##
## data:  Sepal.Length and Species
## F = 138.91, num df = 2.000, denom df = 92.211, p-value < 2.2e-16
```

`aov()`

`aov()` performs ANOVA but is more general purpose and tends to produce output resembling that from other statistics programs. The call to `aov()` is similar to the call to `oneway.test()`.

```
res <- aov(Sepal.Length ~ Species, data = iris)
print(res)
```

```
## Call:
##    aov(formula = Sepal.Length ~ Species, data = iris)
##
## Terms:
##                  Species Residuals
## Sum of Squares  63.21213  38.95620
## Deg. of Freedom        2       147
##
## Residual standard error: 0.5147894
## Estimated effects may be unbalanced
```

```
summary(res)
```

```
##              Df Sum Sq Mean Sq F value Pr(>F)    
## Species       2  63.21  31.606   119.3 <2e-16 ***
## Residuals   147  38.96   0.265                   
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

`lm()`

ANOVA is understood as being a particular instance of a linear model. Linear models will be discussed later, but for now we can see how `lm()`, the primary function for estimating linear models, can be used for estimating the ANOVA model parameters and performing the ANOVA test.

When using `lm()`, the call is `lm(x ~ f - 1, data = d)`.

```
res2 <- lm(Sepal.Length ~ Species - 1, data = iris)
print(res2)
```

```
## 
## Call:
## lm(formula = Sepal.Length ~ Species - 1, data = iris)
## 
## Coefficients:
##     Speciessetosa  Speciesversicolor   Speciesvirginica  
##             5.006              5.936              6.588
```

```
summary(res2)
```

```
## 
## Call:
## lm(formula = Sepal.Length ~ Species - 1, data = iris)
## 
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -1.6880 -0.3285 -0.0060  0.3120  1.3120 
## 
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)    
## Speciessetosa       5.0060     0.0728   68.76   <2e-16 ***
## Speciesversicolor   5.9360     0.0728   81.54   <2e-16 ***
## Speciesvirginica    6.5880     0.0728   90.49   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5148 on 147 degrees of freedom
## Multiple R-squared:  0.9925, Adjusted R-squared:  0.9924
## F-statistic:  6522 on 3 and 147 DF,  p-value: < 2.2e-16
```

We should interpret the results found here as only an estimate of the aforementioned ANOVA model. We should not read anything into the statistical tests performed. The reason why is because effectively all that's being done is testing whether the means of any of the populations are zero, which generally isn't of interest in this context.

The above command estimated the aforementioned ANOVA model verbatim, but different formulations of the ANOVA model exist. For instance, we could say:

$$x_{i1} = \beta_1 + \epsilon_{i1}$$

$$x_{ik} = \beta_1 + \beta_k + \epsilon_{ik}$$

We interpret $\mu_1 = \beta_1$ and $\mu_k = \beta_1 + \beta_k$, or $\beta_k = \mu_k - \mu_1$. We would then rewrite our hypotheses as:

$$H_0 : \beta_2 = \cdots = \beta_K = 0$$

$$H_A : \beta_k \neq 0 \text{ for some } k$$

The `lm()` call `lm(x ~ f, data = d)` estimates the parameters of this model and perform the ANOVA test.

```
res3 <- lm(Sepal.Length ~ Species, data = iris)
print(res3)
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Species, data = iris)
##
## Coefficients:
##       (Intercept)  Speciesversicolor   Speciesvirginica
##             5.006              0.930              1.582
```

```
summary(res3)
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Species, data = iris)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.6880 -0.3285 -0.0060  0.3120  1.3120
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         5.0060     0.0728  68.762  < 2e-16 ***
## Speciesversicolor   0.9300     0.1030   9.033 8.77e-16 ***
## Speciesvirginica    1.5820     0.1030  15.366  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5148 on 147 degrees of freedom
## Multiple R-squared:  0.6187, Adjusted R-squared:  0.6135
## F-statistic: 119.3 on 2 and 147 DF,  p-value: < 2.2e-16
```

Finding the Differences

Rejecting the null hypothesis of no difference provides useful information; we know that at least some of the population means are different. However, we also need to determine *which* means are different, and by how much they differ.

One idea is to compute confidence intervals for differences in means and see which intervals include 0. Any intervals not including zero suggest that the corresponding two populations differ in their means. But if we compute t confidence intervals as we have done before, then we run into the same multiple hypothesis testing problem we had before. Again, we could look to the Bonferroni correction for guidance, but the original problem of being perhaps too conservative still stands.

A less conservative approach is the Tukey honest significant difference approach. With this approach, a single Type I error rate α is chosen to represent the probability of *any* rejection of the null hypothesis of no difference being an error. Then for every pair of populations we compute a confidence interval for the difference in means. We can then use these intervals to determine by how much means of different populations differ.

Recall the object `res` above formed by a call to `aov()`. This object is of class `aov` and the function `TukeyHSD()` can accept it as an argument. `TukeyHSD()` can then compute the desired confidence intervals.

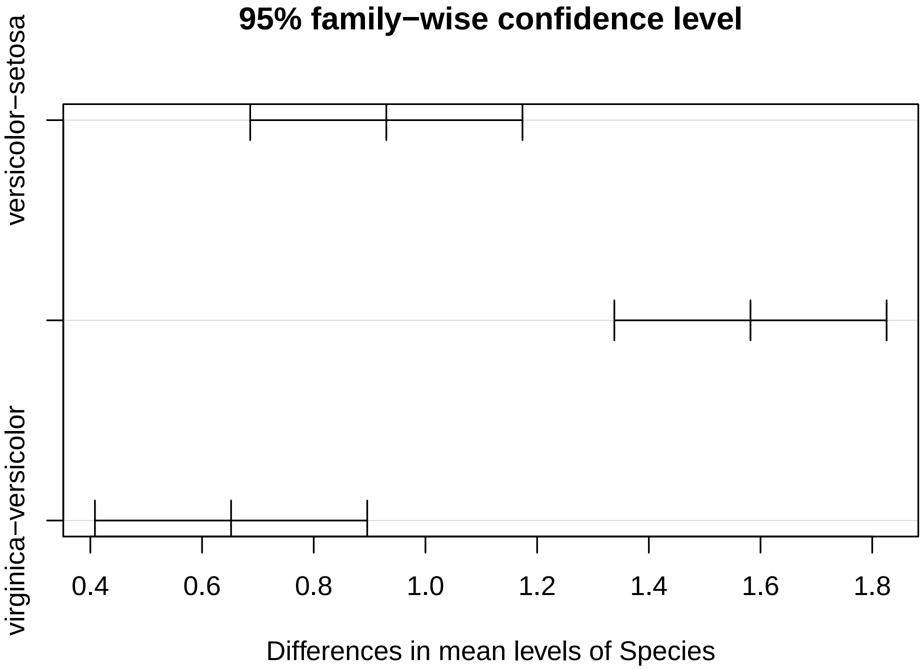
Let's demonstrate:

```
TukeyHSD(res)
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = Sepal.Length ~ Species, data = iris)
##
## $Species
##                       diff       lwr       upr p adj
## versicolor-setosa    0.930 0.6862273 1.1737727     0
## virginica-setosa     1.582 1.3382273 1.8257727     0
## virginica-versicolor 0.652 0.4082273 0.8957727     0
```

The printed output is certainly informative but plots are nice to have. Fortunately plotting these intervals is also easy.

```
plot(TukeyHSD(res))
```



We see that 0 is in none of these intervals. This means that there's significant evidence that every pair of population means are different, with virginica and setosa flowers having the greatest difference in sepal length and versicolor and virginica the least difference.

There are of course parameters we can set if we want, say, different confidence levels. See the function documentation for more details.

Lecture 5

R Package Development

You have been installing and loading packages almost as soon as you learned the basics of R. The strength of R is in its package community, providing packages implementing tools for data cleaning and management, computing algorithms and data structures, and both common and esoteric statistical procedures. Additionally packages are often written by experts in the respective topic (maybe even the procedure's inventor) and the quality of the code is checked and enforced by CRAN, a repository maintained on a volunteer basis. The R language could remain relevant for years–dacades, perhaps–based only on the strength of its package ecosystem (and low price and open source nature).

Today I will introduce you to package development. You may believe that you will never be an R "programmer" and simply will write enough R code to complete the statistical analysis you have been tasked with. While many people reading these lecture notes won't publish a package on CRAN (though you may be surprised!), several reasons exist for *everyone* to learn package development.

As of this writing I am an academic statistician who studies and develops novel statistical methods. This fact alone would justify me personally writing packages: I can distribute the methods I develop to a wider audience. (As of this writing I have one package on CRAN, **CPAT**: a package for change point hypothesis testing, developed as part of recent research projects.) However, even when my research topics are vague, I start writing code and organizing files in anticipation of a package eventually emerging, and even when I'm still conducting research I use the R package structure as the organizational basis of my code and files. That is, I conduct research *via* package development.

Why conduct research via package development? First, package development encourages good coding and project management habits. Good coding habits include:

- Following the DRY principle: don't repeat yourself. This means identifying repeated idioms in code and turning those idioms into generalized functions.
- Sufficiently abstracting code with functions. Abstraction brings two bene-

fits. One, it hides implementation details, making those details *easier* to change rather than harder, since they need to be changed only in a handful of places. Second, your code is easier to read and reason about since the code largely consists of well-named functions with well-defined jobs that code readers can investigate only if they feel the need to do so. If this is done well, then there's no need to extensively comment your code; the code explains itself. Some may even say the code itself *is* the documentation.
- Documenting your code. In package development, this generally means writing documentation for functions that: gives the function a title and description; explains input parameters, what's assumed about them, and how they're used; describing function output; any other important information about how the function works (such as what algorithms are used, potential errors, etc.); and some not-too-complicated examples demonstrating how the function should be used. In my own work, rarely will I put comments in code or function bodies; the function documentation is all that I need. (Also, I set up my text editor, Vim, to build templates for function documentation automatically.)
- Keeping code short. Here, "short" can mean no more than one "screen". That is, an entire body of code can be seen without scrolling. This is not a hard rule; when you feel you must break this rule, do so. But short bodies of code are easier to understand than long bodies of code. Keep code short by hiding execution details in function calls.
- Keeping source files at a manageable size. By source files, I don't mean `.Rmd` files (which is how you've been submitting your code so far) but `.R` files, the original R scripts, which consist *only* of valid R code (though perhaps code for other programs and languages exists in the form of special comments). All functions in a file should be closely related and serve a common purpose. Perhaps one file contains a class definition and related methods, another generic function definitions, another functions implementing a statistical test, another plotting functions, another functions just for data management and cleaning. When splitting code across files, we neither want "tower" files with all the code nor "pancake" files where 1000 files contain only one function each.
- Writing code tests. In programming, errors are great; they tell us immediately something is wrong. Far worse are mistakes that never reveal themselves and thus causing us to sell garbage results. We write code tests to make sure that code functions as we intend it, and so that changes we make don't have unintended consequences. Additionally, we can give our tests to other users for them to read, run, and verify that our code works as intended.
- Flexible code with replicable results. We can modify our code for new contexts and to meet the new demands of our employers/clients, and others can run the same code and get the same results.

Bad coding habits include:

- Copy/paste programming. Sure, it was quick to copy/paste the first time, but if you need to change that original procedure, you now need to change all 1000 instances of that idiom.

- Long "spaghetti" code that only the original author can read (but probably not a few years later after the original project ended and she forgot everything) with things happening many pages prior to when it became interesting, repeats itself to the point of dulling the senses, and littered with the worst kind of comments: the out-of-date comment that never was changed to reflect important changes in the code and thus *misleads* readers.
- Code without any documentation or explanation, so no one knows what it does without careful reading. That takes time.
- Single source files with all the code of a project that gets executed via `source()` for the analysis and thus doesn't allow for easy modification. A small change near the end of the file would mean the entire project needs to be re-run, which takes hours. That or someone opens the file and carefully does just the part that needs to change, which invites errors.
- Untested code that gets changed and suddenly produces subtle bugs that takes days to diagnose when a problem should have been flagged much earlier and closer to the original violated assumption.
- Code that's so rigid and convoluted you don't dare change it and risk breaking its delicate structure, even when the client just wants one additional statistical test done.

Remember, kids: computer scientists develope the standards they did and the tools they use for good reasons that apply beyond software development. Ignore them at your own peril.

Second, you may discover that code initially written for one specific task is actually generally useful. Both yourself and others can take advantage of all the work you did in the future and save time. Furthermore, this code was put in a package, which means its easily accessed!

Third, package development can be an important part of **reproducible research**. You may have heard that reproducibility is an important part of scientific research. Here, we want to be sure that the original analysis itself can be reproduced by anyone with the code and the same results can be obtained. This can allow others to diagnose any problems in our work (or verify that the work was done correctly) and perhaps modify, say, the input data, to replicate the research project and ensure that the results are robust.

I personally use package development as an important part of my own research pipeline. I combine it with a Linux/Unix development environment (turns out the black screen of death is **extremely useful** when you take the time to learn it, far more useful than GUIs), executable shell scripts (I use R Markdown or LaTeX only for writing documentation or presenting results, *not* for actually obtaining results), GNU **make** for defining file dependencies and tracking what has changed and what needs to be changed (bringing my project up to date after making changes is as easy as typing `make` at the command line, and anything that needs to be changed will be changed while the rest is untouched), and of course version control. (I won't be teaching version control in this class but you really should learn some tool, such as **git**. Did you make a change that broke everything? Should have used version control. Experiments with your code didn't pan out? Should have used version control. Delete important lines or even an entire file? Should have used version control. Your folder is filled with files with minor name changes just for trying out little changes? Should have used version control.)

You may think you won't ever work on a project that gets big enough to warrant a package. Well, projects have a tendency to get bigger and more involved that initially thought. One day you'll open your laptop and discover that the "quick project" you were doing has several associated files and a single script thousands of lines long. Then you'll wish you had a package to manage all this code. In short: do it right at the beginning. You'll save yourself trouble.

Starting a Package in RStudio

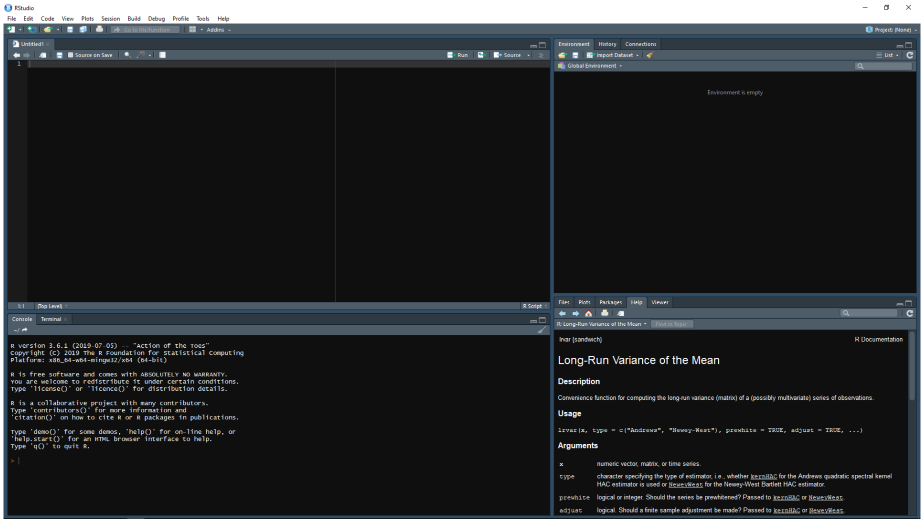
I personally do not use RStudio for writing packages. I write my code (including these lecture notes) in a text editor called Vim. When I want to make a package, I create the necessary files and folders manually, and when I need to build the package I go to the "black screen of death" and type the necessary commands myself. But many people like GUIs so I will show how to use RStudio for starting a package, managing files, and building the package.

You may need to install additional software to get started. Specifically, you may need GNU software development tools (in particular a C/C++ compiler) and a LaTeX installation. Getting these tools depends on your platform (Windows, Mac OS X, Unix/Linux), so prior to proceeding visit this document for further instructions. (You shouldn't need to pay for anything.) Additionally there are two packages you should install that help with the development process: **devtools** and **roxygen2**. The former is a toolbox for development and you may even already have it installed to facilitate installing packages from GitHub. **roxygen2** allows for writing documentation by recognizing certain comments with special syntax as documentation. While you're at it, install **testthat**; it's the package we will use for writing tests, though we won't discuss tests in the current lecture.

Okay, let's start making a package. Recall the code we wrote implementing a class representing a financial account a few lectures ago. Let's turn that code into a package, calling the package **account**. When we start RStudio we may have a screen resembling the following:

Open the `File` menu and click `New Project`.

Projects have a rigid directory structure and care a great deal about what files are in what folders. Since we have a fresh new package we want a fresh new



```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

lrvar {sandwich} R Documentation

Long-Run Variance of the Mean

Description

Convenience function for computing the long-run variance (matrix) of a (possibly multivariate) series of observations.

Usage

```
lrvar(x, type = c("Andrews", "Newey-West"), prewhite = TRUE, adjust = TRUE, ...)
```

Arguments

`x`	numeric vector, matrix, or time series.
`type`	character specifying the type of estimator, i.e., whether `kernHAC` for the Andrews quadratic spectral kernel HAC estimator is used or `NeweyWest` for the Newey-West Bartlett HAC estimator.
`prewhite`	logical or integer. Should the series be prewhitened? Passed to `kernHAC` or `NeweyWest`.
`adjust`	logical. Should a finite sample adjustment be made? Passed to `kernHAC` or `NeweyWest`.

Figure 1: RStudio start

directory. Click `New Directory` in the pop-up window.

When asked what type of project we're starting, click `R Package`.

We should now see in our pop-up window queries about package details. Name the package "account". If you want you can choose which directory the new directory should be placed in.

The window should close and now RStudio should resemble the screen below.

RStudio created a basic package with minimal structure. The most essential files and folders have been made, and we will be modifying them. RStudio even gave us a starting R file with some tips.

Additionally the directory RStudio created looks like this:

The file `hello.R` RStudio created is in the `R/` subdirectory. I will explain package subdirectories more later. Let's first check that our package can be built and installed by pressing (on Windows/Linux) `Ctrl+Shift+B`. When you do you should see output in one of the RStudio panes resembling the following:

Additionally, in the R console, you should see `library(account)` run. If all went smoothly, you should be able to type `hello()` in the console and have R respond with `[1] "Hello, world!"`.

What just happened is RStudio executed programs (specifically `R CMD build` and `R CMD install`) that built and installed the package **account**, then restarted R and loaded in the package. Since the package is currently extremely minimal, there should have been no problem. That doesn't mean there's no problems,

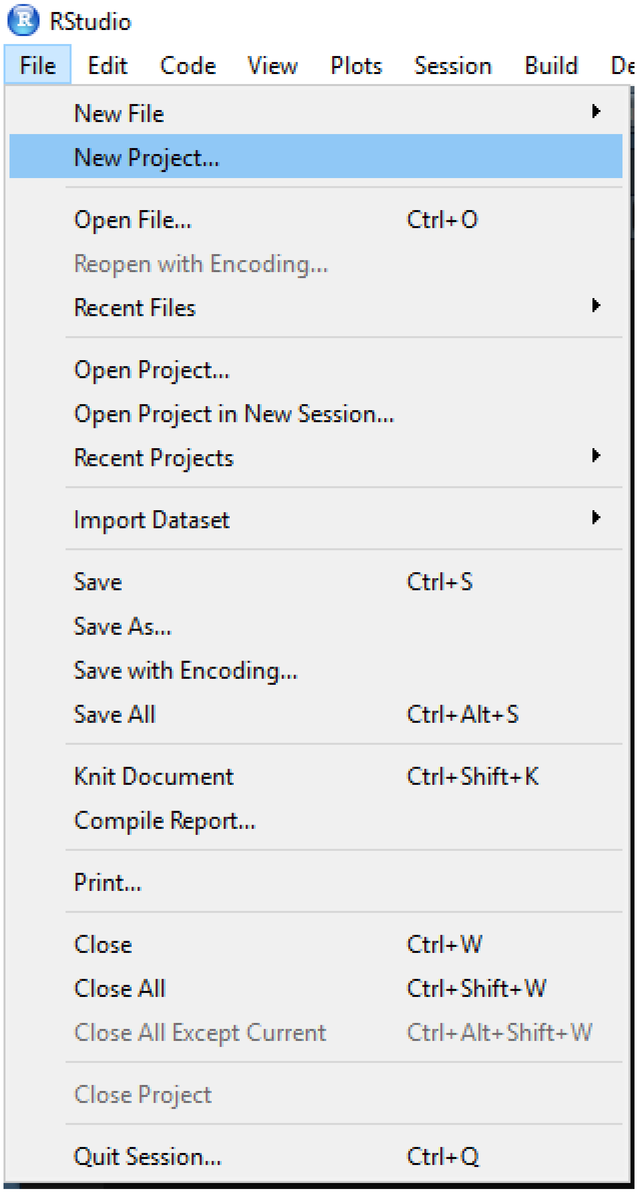
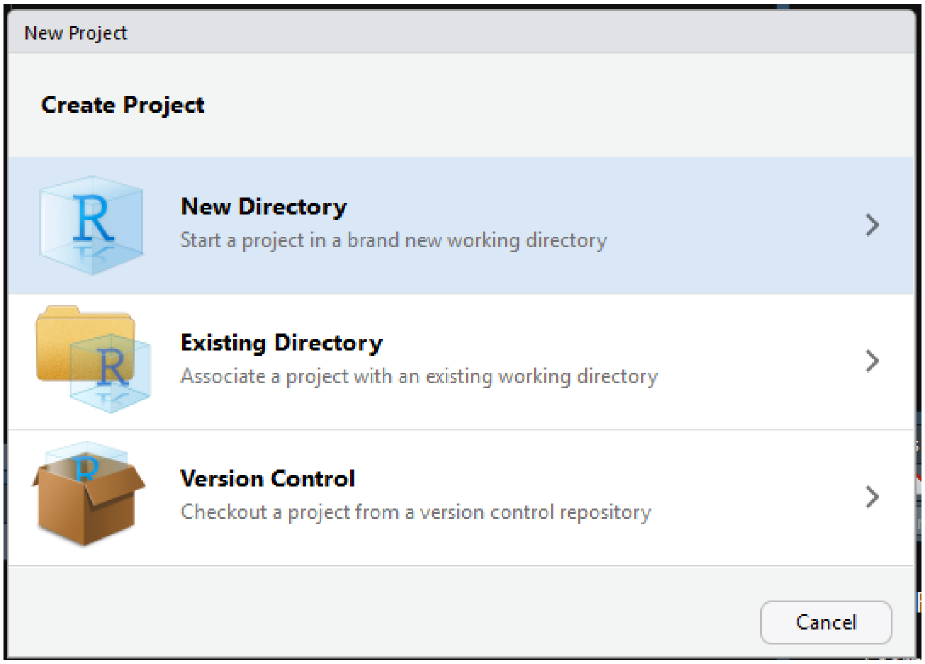


Figure 2: Click File->New Project



New Project

Create Project

New Directory
Start a project in a brand new working directory

Existing Directory
Associate a project with an existing working directory

Version Control
Checkout a project from a version control repository

Figure 3: Click New Directory

though. In fact, in its current form, this package would certainly not be accepted to CRAN. This isn't just because the package doesn't really do anything.

Press `Ctrl+Shift+E` to test the package. RStudio then runs `R CMD check`, which will run various tests on the package to make sure that it adheres to selected standards and appears to be an acceptably functional package. We get output resembling the following:

Near the bottom we should see the following:

In this run, there was a warning and a note. When checking packages, there are three types of flags: errors, warnings, and notes. Errors signify a problem that will cause the package to fail to build or install. Warnings appear for issues that won't necessarily cause the package to fail to build or install but are considered bad practice and need to be addressed. Finally, notes signify issues less serious than warnings; they call attention to those issues. Sometimes notes will be thrown just because a package is new.

If we were thinking of submitting our package to CRAN, there would have to be no errors or warnings at all when the package is checked via `R CMD check --as-cran` (which is a check with more tests to see the package meets CRAN standards). While there can be notes they need to be brought to a minimum, and you may need to defend your package in the presence of the note to the

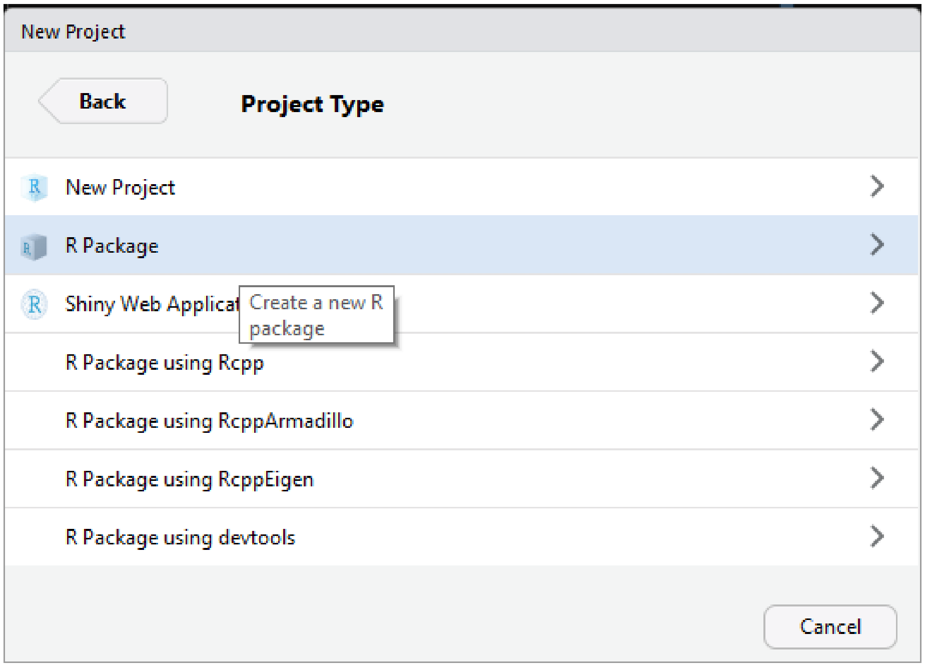


Figure 4: Click R Package

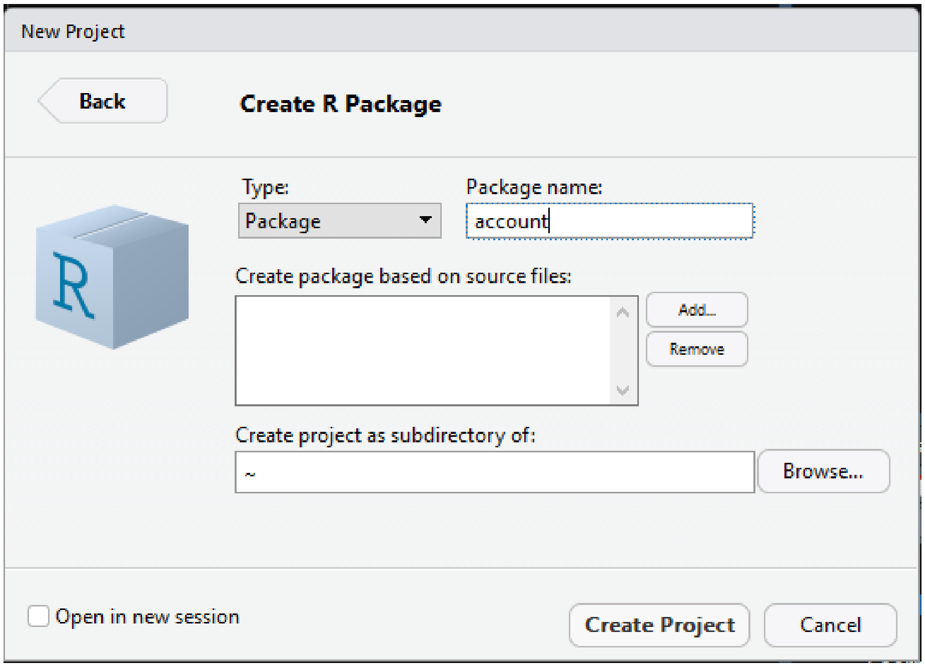
CRAN volunteer checking the submission. In general, though, get in the habit of dealing with errors, warnings, and notes immediately and to the best of your ability. Eliminate all errors and warnings and think about how to handle the notes.

In this case we see that a warning was thrown because `R CMD check` could not understand the package license. (The note is more mysterious; it could have been an issue local to the circumstances of my writing these notes at my grandparents' house in rural Idaho without Internet access. We can probably ignore it.) This brings us to the first important package file: `DESCRIPTION`.

DESCRIPTION

`DESCRIPTION` is simply a structured plain text file with the package's metadata. This include the package's name, description, version, author, maintainer, and dependencies. Below is the `DESCRIPTION` file generated by RStudio:

```
Package: account
Type: Package
Title: What the Package Does (Title Case)
Version: 0.1.0
Author: Who wrote it
```



Type: Package

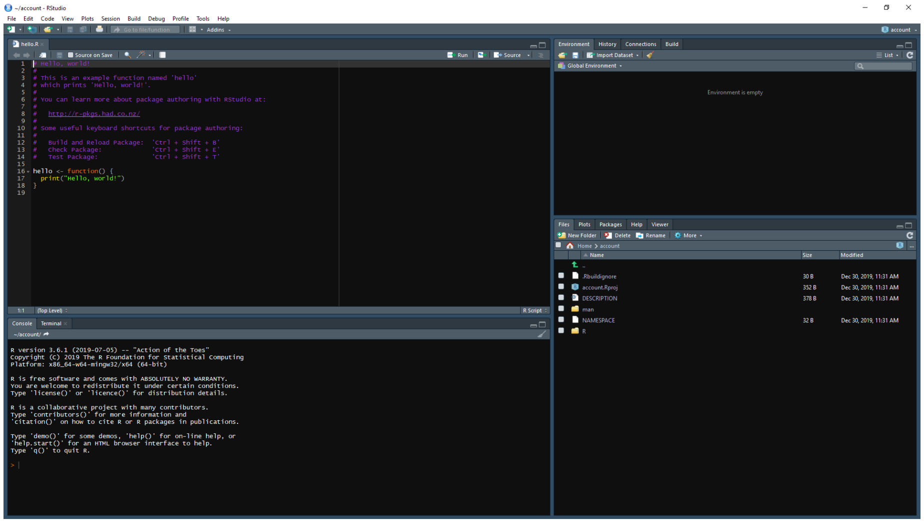
Package name: account

Create package based on source files:

Create project as subdirectory of: ~

☐ Open in new session

Figure 5: Name the package



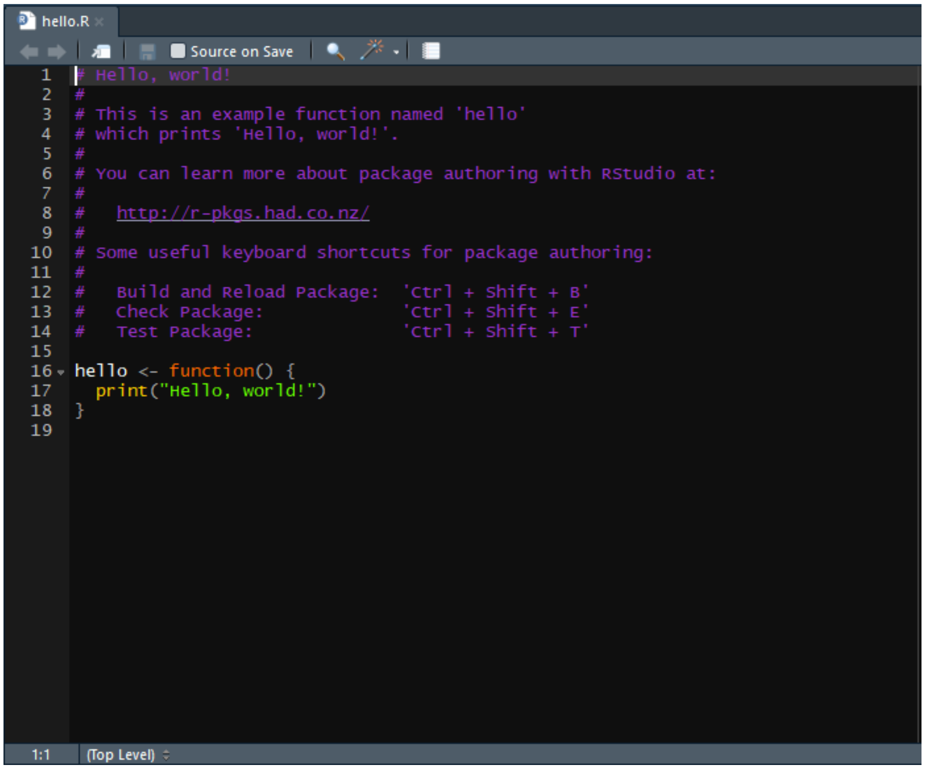
```r
# Hello, world!
#
# This is an example function named 'hello'
# which prints 'Hello, world!'.
#
# You can learn more about package authoring with RStudio at:
#
#   http://r-pkgs.had.co.nz/
#
# Some useful keyboard shortcuts for package authoring:
#
#   Build and Reload Package:  'Ctrl + Shift + B'
#   Check Package:             'Ctrl + Shift + E'
#   Test Package:              'Ctrl + Shift + T'

hello <- function() {
  print("Hello, world!")
}
```

```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Environment is empty

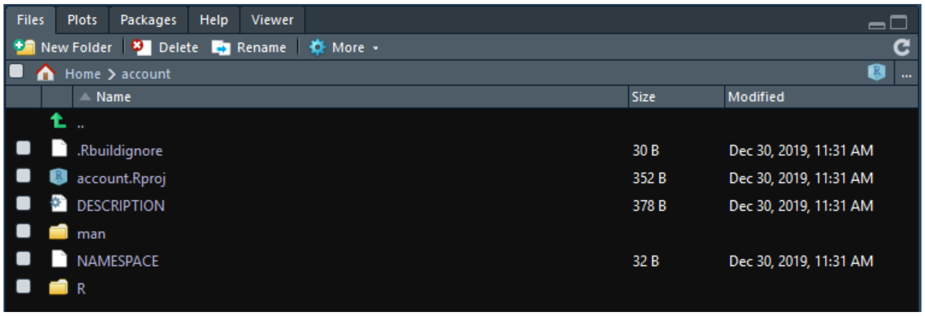
Name	Size	Modified
..		
.Rbuildignore	30 B	Dec 30, 2019, 11:31 AM
account.Rproj	352 B	Dec 30, 2019, 11:31 AM
DESCRIPTION	378 B	Dec 30, 2019, 11:31 AM
man		
NAMESPACE	32 B	Dec 30, 2019, 11:31 AM
R		

Figure 6: New RStudio screen



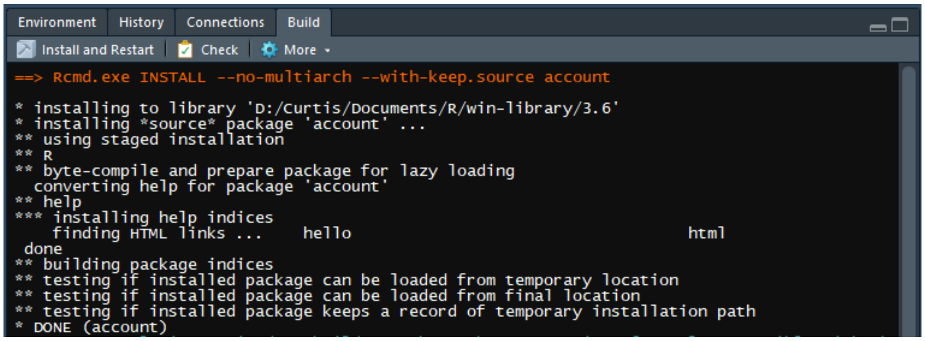
```r
# Hello, world!
#
# This is an example function named 'hello'
# which prints 'Hello, world!'.
#
# You can learn more about package authoring with RStudio at:
#
#   http://r-pkgs.had.co.nz/
#
# Some useful keyboard shortcuts for package authoring:
#
#   Build and Reload Package:  'Ctrl + Shift + B'
#   Check Package:             'Ctrl + Shift + E'
#   Test Package:              'Ctrl + Shift + T'

hello <- function() {
  print("Hello, world!")
}

```

Figure 7: New R file



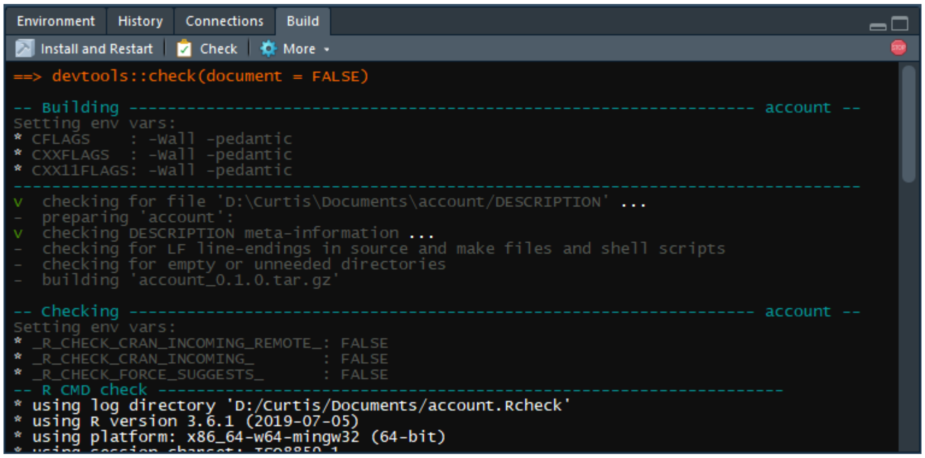
Name	Size	Modified
..		
.Rbuildignore	30 B	Dec 30, 2019, 11:31 AM
account.Rproj	352 B	Dec 30, 2019, 11:31 AM
DESCRIPTION	378 B	Dec 30, 2019, 11:31 AM
man		
NAMESPACE	32 B	Dec 30, 2019, 11:31 AM
R		

Figure 8: Package directory



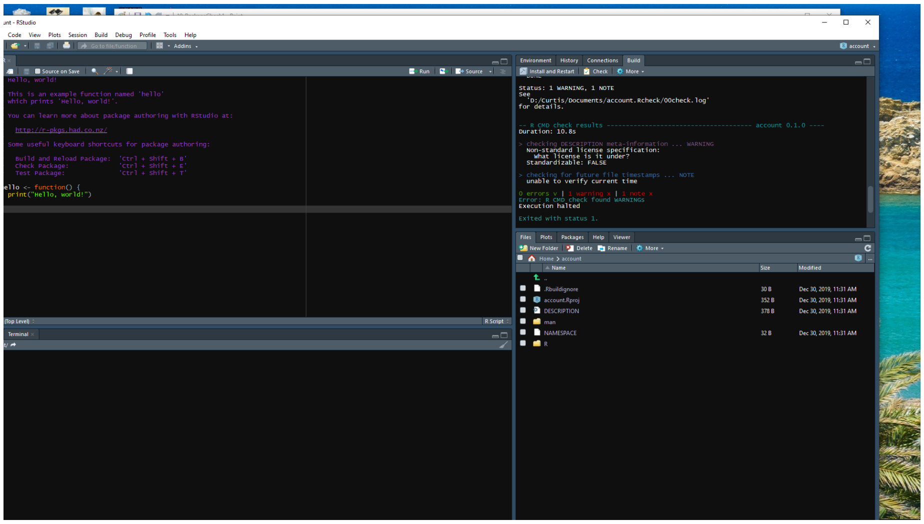
```
==> Rcmd.exe INSTALL --no-multiarch --with-keep.source account

* installing to library 'D:/Curtis/Documents/R/win-library/3.6'
* installing *source* package 'account' ...
** using staged installation
** R
** byte-compile and prepare package for lazy loading
  converting help for package 'account'
** help
*** installing help indices
    finding HTML links ...    hello                                   html
 done
** building package indices
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (account)
```

Figure 9: Package build



```
==> devtools::check(document = FALSE)

-- Building ----------------------------------------------------------------- account --
Setting env vars:
* CFLAGS    : -Wall -pedantic
* CXXFLAGS  : -Wall -pedantic
* CXX11FLAGS: -Wall -pedantic
----------------------------------------------------------------------------------------
v  checking for file 'D:\Curtis\Documents\account/DESCRIPTION' ...
-  preparing 'account':
v  checking DESCRIPTION meta-information ...
-  checking for LF line-endings in source and make files and shell scripts
-  checking for empty or unneeded directories
-  building 'account_0.1.0.tar.gz'

-- Checking ----------------------------------------------------------------- account --
Setting env vars:
* _R_CHECK_CRAN_INCOMING_REMOTE_: FALSE
* _R_CHECK_CRAN_INCOMING_       : FALSE
* _R_CHECK_FORCE_SUGGESTS_      : FALSE
-- R CMD check ---------------------------------------------------------------
* using log directory 'D:/Curtis/Documents/account.Rcheck'
* using R version 3.6.1 (2019-07-05)
* using platform: x86_64-w64-mingw32 (64-bit)
```

Figure 10: Package check


unt - RStudio
Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
account
Source on Save
Run
Source
Environment History Connections Build
Install and Restart
Check
More
Files Plots Packages Help Viewer
New Folder Delete Rename More
Home > account
(Top Level)
R Script
Terminal

```
Hello, world!

This is an example function named 'hello'
which prints 'Hello, world!'.

You can learn more about package authoring with RStudio at:

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

Some useful keyboard shortcuts for package authoring:

  Build and Reload Package:  'Ctrl + Shift + B'
  Check Package:             'Ctrl + Shift + E'
  Test Package:              'Ctrl + Shift + T'

ello <- function() {
 print("Hello, world!")
```

```
Status: 1 WARNING, 1 NOTE
See
  'D:/Curtis/Documents/account.Rcheck/00check.log'
for details.


-- R CMD check results ------------------------------------ account 0.1.0 ----
Duration: 10.8s

> checking DESCRIPTION meta-information ... WARNING
  Non-standard license specification:
    What license is it under?
  Standardizable: FALSE

> checking for future file timestamps ... NOTE
  unable to verify current time

0 errors v | 1 warning x | 1 note x
Error: R CMD check found WARNINGS
Execution halted

Exited with status 1.
```

Name	Size	Modified
..		
.Rbuildignore	30 B	Dec 30, 2019, 11:31 AM
account.Rproj	352 B	Dec 30, 2019, 11:31 AM
DESCRIPTION	378 B	Dec 30, 2019, 11:31 AM
man		
NAMESPACE	32 B	Dec 30, 2019, 11:31 AM
R		

Figure 11: Package problems

```
Maintainer: The package maintainer <yourself@somewhere.net>
Description: More about what it does (maybe more than one line)
    Use four spaces when indenting paragraphs within the Description.
License: What license is it under?
Encoding: UTF-8
LazyData: true
```

Many of these fields are self-explanatory and we can easily fill them out. Some fields we may never need to edit (such as `Encoding` and `LazyData`). That said, let's explain some of these fields:

- `Package`: The name of the package. Make it short and sensible. Bonus points if there's some logic to the naming scheme you use for packages (such as all lower case, all upper case, lowerCamelCase, UpperCamelCase, etc.; programmers fight holy wars over naming conventions).
- `Type`: This is an R package.
- `Title`: The title of the package (a short description, not even a sentence).
- `Version`: The package version number. Version numbering matters and signals a lot about the state and stability of the package, so here are some conventions. When the package is initially born, the version number should be `0.0.0.9000`. The first number in the version number indicates "major version" level, the second "minor version" level, and third "patch version" level. In this case we threw on a fourth number, `9000`, as a big number to get readers' attention. This number alerts any reader that this version of the package is currently under development and could change on even a daily basis; it's as unstable as a package can get. When we are no longer developing this version, we will drop the `9000` and simply use

version `0.0.1` or `0.1.0`, depending on our opinion on how developed the package is. The former number would signal that the package is in **alpha** development stage; important features in the package are missing and most of its features should not be considered stable. The latter indicates a **beta** development stage; the package can be safely used and the features there are considered somewhat stable, but the package is missing important features that should eventually be put in. A `1.0.0` release would be a **stable** release; all planned features are present and the interface is stable. As we update our package we would increment each of these three numbers depending on how significant the changes made were. We would change the last number for minor patches and bug fixes. Changing the middle number signifies noticeable changes in the package's functionality, but nothing drastic. Changing the first number indicates major changes in the package's functionality, perhaps even breaking backwards compatibility.
- `Author`: Give yourself credit! Also, provide an e-mail in angle brackets, like `Joe Diamond <joe@arkhampi.com>`.
- `Maintainer`: The initial author of the package may not be the current maintainer of the package (the person fixing bugs and releasing updates), but often the author and maintainer lines have a lot of overlap.
- `Description`: What does the package provide? What does it do? Why is it here? Put that information here.
- `License`: Under what legal license is the package released under? Under what legal conditions may others use, copy, or distribute this package? Is this package proprietary or free and open source? If this is a personal package that you don't plan on releasing to the world, you're welcome to ignore this field. But you probably shouldn't, though; if you don't provide a license then it's harder for you yourself to safely share your package with others. R recognizes a number of values for this field, including common open source licenses (`GPL-2`, `GPL-3`, `MIT`, `CC0`) and also the value `file LICENSE` (or in the case of the MIT license, `MIT + file LICENSE`, since `MIT` is merely a license template). If `file LICENSE` is used, R will expect a plain text file in the base directory of the package called `LICENSE`, containing the licensing information. If your package is proprietary, you would use this option. Note that CRAN needs some open source license in order to distribute your package.
- `Encoding`: The file encoding. You'll edit this if you know what you're doing.
- `LazyData`: Should data be lazy loaded? See above.

Some additional fields not shown above but often appearing in packages:

- `Imports`: The other packages this package *must* have in order to function. We will be leaving this field blank for now, but below is an example of how it's used:

```
Imports:
    dplyr,
    ggplot2,
    data.table
```

- `Depends`: Dependencies of the package. We should put at minimum `R (>= current.version.number)`, but it's possible that packages will be listed here in a manner similar to how they were listed in `Imports`. The difference between listing a package under `Depends` and listing one under `Imports` is that a package listed under `Depends` will also be loaded to the namespace via a command like `library()` in addition to our package. `Imports`, in comparison, does not attach the other package to the namespace but loads without attaching. (The package is available but users can't access its content without `::`.)
- `Suggests`: Similar to `Imports`, but the packages listed are not *essential*; the package may work better if these are present, though.
- `VignetteBuilder`: Packages that build vignettes (which we will discuss later).
- `RdMacros`: Packages with macros for `.Rd` files. See `Encoding` above. (I only mention it because the packages I used for citation management require editing this field.)

Many other fields for the `DESCRIPTION` file exist, though they don't need to be present in order for the file to be valid. Look them up if you need them.

We already have enough information to update our `DESCRIPTION` file, though. Here's the new `DESCRIPTION`; copy and paste the text below to replace the old file.

```
Package: account
Type: Package
Title: Data Structures for Managing Banking Accounts
Version: 0.1.0
Author: Curtis Miller <cmiller@math.utah.edu>
Maintainer: Curtis Miller <cmiller@math.utah.edu>
Description: This package implements an S3 class for representing and managing
    bank accounts, with accompanying helper functions and methods.
Depends: R (>= 3.6.0)
License: CC0
Encoding: UTF-8
LazyData: true
```

Other Files in the Base Directory

I next describe other important files in the package's base directory. All of these are plain text files. `NAMESPACE` and `.Rbuildignore` are essential, while others don't *need* to be present, though including them is considered good practice.

`NAMESPACE`

The `NAMESPACE` file is a plain text file with content resembling R code. This file controls what objects or packages from other packages are imported and what the package should make available to users loading the package via `library()`. The file defines what objects are **public** (available on package attachment or via `::`) and what objects are **private** (used only internally in the package, not intended to be used by package users, and can be used only via `:::`).

If you are using **roxygen2** you may not need to edit this file yourself. In this lecture we won't use **roxygen2**. That said, we will edit this file later.

`.Rbuildignore`

This file lists regular expression (regex) patterns identifying files that R should ignore when building the package. R will assume that every file in the package directory matters to the package. If it doesn't recognize a file, an error will be thrown and the package won't build. The answer though isn't that we shouldn't put irrelevant files in the package directory, but instead that we should inform R that these files need to be ignored during package building. So we list those files here.

RStudio populated this file with the following:

```
^.*\.Rproj$
^\.Rproj\.user$
```

The first line says ignore any file that ends with `.Rproj` (an RStudio project file) and the second to ignore the file `.Rproj.user` in the base directory. These are regular expressions. Many resources exist for learning regular expressions but for now I will give you two basic ways to add files:

- If you want to list a specific file to be ignored, include the line `^filename$`. You *must* include `^` at the beginning and `$` at the end. `^` in regex means "beginning of line" and `$` means "end of line". Failing to include these could lead to any file with the string `filename` in its path being matched and thus excluded from the build. Above `filename` is assumed to be in the base directory of the package; if you want to list a file in a subdirectory, use `^path/to/filename$` instead (always use `/` for file paths, even on Windows systems, where `\\` is customary). If your file name include a `.` (even if it's to separate an extension from the main file name), escape the `.` with a backslash, like so: `^filename\.ext$` to match file `filename.ext`. (`.` has a special meaning in regex, meaning "zero or more instances of".)
- If you want to exclude an entire directory, use `^directory/` or `^path/to/directory/`.
- If you want to exclude all files with a certain extension, use `^.*\.ext$`, where `ext` is the extension you want to exclude (such as `txt` or `csv`). In regex, `.*` means "match zero or more things", and `\.` means a literal

period. This line will match files not just in the base directory but any subdirectory in the package.

For more nuanced patterns, study regex, and realize that the string being matched is the entire file path beyond the package directory.

We will be adding more files to the package directory we want ignored, so we will add the following lines to `.Rbuildignore`:

```
^README\.md$
^NEWS\.md$
```

`README.md`

This file is a Markdown file describing the package, going into more depth than what was written in `DESCRIPTION`. This file is meant to be read by humans, and it's formatted using Markdown syntax. (R Markdown is a special flavor of Markdown; if you can write documents in R Markdown, you can write Markdown files.) This file is meant for new users. In `README.md` I recommend:

- Describing the package, what it's for, and why people should use it;
- How people can get and install the package (and any prerequisites); and
- How people should use the package, including perhaps example code.

`README.md` should not be too long and any examples included in it as simple as possible. This will be the `README.md` file of our package:

~~~
# account
*Version 0.1.0*

**account** is a package providing functions and class definitions for handling
accounts resembling traditional banking accounts. If installing from the base
directory of the package and the **devtools** package is installed, **account**
can be installed using `devtools::install()`.

Below are examples of using **account**.

```r
library(account)

# Creating an account
my_account <- account(title = "My Personal Account",
                      owner = "Joe Diamond",
                      start = "1925-01-01",
                      init = 1000)
# Adding a transaction to the account
my_account <- my_account + account_transaction("1925-01-02",
                                               -10,
                                               "Ammo for guns")
~~~

~~~
print(my_account)

# Example account object
accdemo
summary(accdemo)
plot(accdemo)
```
~~~

`NEWS.md`

This file is a Markdown document intended for existing users of the package. This file tracks changes to the package made with each version of the package, such as any functions that were changed or implementation changes made between versions, along with any bug fixes.

Our package is brand new so there's not much news. We will get `NEWS.md` started with the following:

```
# account News

## Version 0.1.0
*2019-12-31*
---

- Package **account** created
- S3 classes `account`, `transaction`, and `summary.account` with associated
  constructor functions defined
- `print()`, `summary()`, `sort()`, `plot()`, and `+` methods for `account`
  objects created
- Generic function `as.account()` created
- `print()` and `as.account()` methods for `transaction` objects created
- `print()` method for `summary.account` objects created
- `account` helper functions `account_title()`, `account_owner()`,
  `account_transactions()`, with associated assignment versions (such as
  `account_title<-()`) created
- `account` helper functions `bad_Initial()`, `all_transactions()`,
  `account_dates()`, `account_trans_amounts()`, `account_memos()`,
  `transaction_count()`, `account_new_transaction()`,
  `account_delete_transaction()`, and `is.account()` created
- `transaction` helper functions `transaction_date()`, `amount()`, `memo()`,
  with associated assignment versions (such as `amount<-()`) created
- `transaction` helper function `is.transaction()` created
- Function `read_csv_account()` created
- Example `account` object `accdemo` created
```

Package Directory Structure

R packages use a well-defined directory structure, with files in certain files being used in specific ways. Below are descriptions of the directories and what goes in them.

`R/`

The `R/` directory is the most important directory of the package. It contains the code for all the R objects you're providing the users of your package. The files in the directory should all be `.R` files (that is, R source files, or "scripts", even though these scripts should do nothing other than create objects and maybe document them).

If we wanted we could put all our package code in a single `.R` file, we could. In fact, if all our package does is provide a couple functions, this may be a perfectly reasonable organization. But if your package is more involved, you should spread your code out over multiple files, where the code in a file serves a common purpose. The order in which these files are loaded should not matter. If it does, something is terribly wrong.

RStudio go you started with the file `hello.R` in the `R/` directory, listed below:

```
# Hello, world!
#
# This is an example function named 'hello'
# which prints 'Hello, world!'.
#
# You can learn more about package authoring with RStudio at:
#
#   http://r-pkgs.had.co.nz/
#
# Some useful keyboard shortcuts for package authoring:
#
#   Build and Reload Package:  'Ctrl + Shift + B'
#   Check Package:             'Ctrl + Shift + E'
#   Test Package:              'Ctrl + Shift + T'

hello <- function() {
  print("Hello, world!")
}
```

You're expected to delete this file and put something useful in the `R/` directory. In our case, we will take the functions we wrote for modeling banking accounts from a previous lecture (plus a couple others to round the package out). The files and their code are listed below. I've chosen to organize the files around the classes and function augmenting them. Copy them verbatim.

transaction.R

This file includes `transaction()`, the function for making `transaction`-class objects, along with associated methods.

```r
transaction <- function(date, amount, memo = "") {
  obj <- list(date = as.Date(date),
              amount = as.numeric(amount),
              memo = as.character(memo)
  )
  class(obj) <- "transaction"
  obj
}

is.transaction <- function(x) {class(x) == "transaction"}

print.transaction <- function(x, space = 10) {
  tdate <- as.character(x$date)
  datestring <- paste0("    ", tdate, ":")
  formatstring <- paste0("%+", space[[1]], ".2f")  # See sprintf() to explain
  amountstring <- sprintf(formatstring, x$amount)
  if (x$memo == "") {
    memostring <- ""
  } else {
    memostring <- paste0("(", x$memo, ")")
  }
  cat(datestring, amountstring, memostring, "\n")
}

as.account.transaction <- function(x, title = "Transaction", owner = "noone") {
  res <- account(start = transaction_date(x), init = amount(x), owner = owner,
                 title = title)
  memo(account_transactions(res)[[1]]) <- memo(x)
  res
}
```

transactionHelpers.R

This file includes functions that are meant to work with `transaction`-class objects.

```r
transaction_date <- function(trns) {
  trns$date
}

`transaction_date<-` <- function(trns, value) {
  trns$date <- as.Date(value)
  trns
}

amount <- function(trns) {
  trns$amount
}

`amount<-` <- function(trns, value) {
  trns$amount <- as.numeric(value)
  trns
}

memo <- function(trns) {
  trns$memo
}

`memo<-` <- function(trns, value) {
  trns$memo <- as.character(value)
  trns
}
```

`account.R`

This file includes `account()`, the function for making `account`-class objects, along with associated methods.

```
account <- function(start, owner, init = 0, title = "Account") {
  obj <- list(
    title = as.character(title),
    owner = as.character(owner),
    transactions = list(transaction(start, init, "Initial"))
  )
  class(obj) <- "account"
  obj
}

is.account <- function(x) {class(x) == "account"}

sort.account <- function(x, decreasing = FALSE, ...) {
  # There might be multiple entries with memo Initial; design code for that
  memo_Initial <- which(account_memos(x) == "Initial")
  if (length(memo_Initial) == 0) {
    date_Initial <- min(account_dates(x))
    true_Initial <- which.min(account_dates(x))
  } else {
    date_Initial <- account_dates(x)[memo_Initial]
    true_Initial <- which((account_dates(x) == min(date_Initial)) &
                          (account_memos(x) == "Initial"))
  }
  tcount <- transaction_count(x)
  nix <- (1:tcount)[-true_Initial]
  ordered_nix <- nix[order(account_dates(x)[nix], decreasing = decreasing, ...)]
  if (decreasing) {
    final_order <- c(ordered_nix, true_Initial)
  } else {
    final_order <- c(true_Initial, ordered_nix)
  }
  account_transactions(x) <- account_transactions(x)[final_order]
  x
}

print.account <- function(x, presort = FALSE) {
  if (presort) {
    x <- sort(x)
  }
  cat("Title:", account_title(x))
  cat("\nOwner:", account_owner(x))
  cat("\nTransactions:\n-----------------------------------------------------\n")
  for (t in account_transactions(x)) {
    print(t)
  }
}

`+.account` <- function(x, y) {
  account_transactions(x) <- c(account_transactions(x), account_transactions(y))
  sort(x)
}

plot.account <- function(x, y, ...) {
  if (bad_Initial(x)) stop("Malformed account object")
  x <- sort(x)
  unique_dates <- unique(account_dates(x))
  date_trans_sum <- sapply(unique_dates, function(d) {
    idx <- which(account_dates(x) == d)
    sum(account_trans_amounts(x)[idx])
  })
  plot(unique_dates, cumsum(date_trans_sum), type = "l",
       main = account_title(x), xlab = "Date", ylab = "Balance", ...)
}
```

accountHelpers.R

This file includes functions that are meant to work with `account`-class objects.

```
account_title <- function(account) {
  account$title
}

`account_title<-` <- function(account, value) {
  account$title <- as.character(value)
  account
}

account_owner <- function(account) {
  account$owner
}

`account_owner<-` <- function(account, value) {
  account$owner <- as.character(value)
  account
}

account_transactions <- function(account) {
  account$transactions
}

`account_transactions<-` <- function(account, value) {
  account$transactions <- value
  account
}

all_transactions <- function(account) {
  all(sapply(account_transactions(account), is.transaction))
}

account_dates <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  Reduce(c, lapply(account_transactions(account), transaction_date))
}

account_trans_amounts <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  sapply(account_transactions(account), amount)
}

account_memos <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  sapply(account_transactions(account), memo)
}

transaction_count <- function(account) {
  length(account_transactions(account))
}

bad_Initial <- function(account) {
  if (!all_transactions(account)) stop("Not all transactions of right class")
  if (!("Initial" %in% account_memos(account))) stop("No Initial transaction")

  memo_Initial <- which(account_memos(account) == "Initial")
  date_Initial <- min(account_dates(account)[memo_Initial])
  sort(account_dates(account))[1] < date_Initial
}

account_new_transaction <- function(...) {
  as.account(transaction(...))
}

account_delete_transaction <- function(account, date = NULL, memo = NULL) {
  if (is.null(date) && is.null(memo)) {
    stop("Must specify at least one of date or memo")
  }

  if (!is.null(date)) {
    filter_dates <- which(account_dates(account) == as.Date(date))
  } else {
    filter_dates <- 1:transaction_count(account)
  }

  if (!is.null(memo)) {
    filter_memos <- which(account_memos(account) == memo)
  } else {
    filter_memos <- 1:transaction_count(account)
  }

  final_filter <- intersect(filter_dates, filter_memos)
  if (length(final_filter) == 0) {
    warning("No transactions with date/memo combination")
    return(account)
  } else {
    account_transactions(account)[final_filter] <- NULL
    return(account)
  }
}
```

summary.account.R

This file includes `summary.account()`, the function for making `summary.account`-class objects, along with associated methods. (I treat `summary.account()` as both a method of `summary()` and as a constructor of an object.)

```
summary.account <- function(object, recent = 5) {
  if (bad_Initial(object)) warning("account object malformed!")
  res <- list()
  res$title <- account_title(object)
  res$owner <- account_owner(object)
  res$balance <- sum(account_trans_amounts(object))
  res$tcount <- transaction_count(object)
  res$rtrans <- account_transactions(sort(object,
                                          decreasing = TRUE)
                                    )[1:min(recent, res$tcount)]

  class(res) <- "summary.account"
  res
}

print.summary.account <- function(x, prefix = "\t") {
  cat("\n")
  cat(strwrap(x$title, prefix = prefix), sep = "\n")
  cat("\n")
  cat("Owner:  ", sprintf("%20s", x$owner), "\n", sep = "")
  cat("Transactions:  ", sprintf("%13s", x$tcount), "\n", sep = "")
  cat("Balance:  ", sprintf("%18.2f", x$balance), "\n", sep = "")
  cat("\nRecent Transactions:\n----------------------------------------------------\n")
  for (t in x$rtrans) {
    print(t)
  }
}
```

Generics.R

This file defines generic functions, along with any methods for external classes.

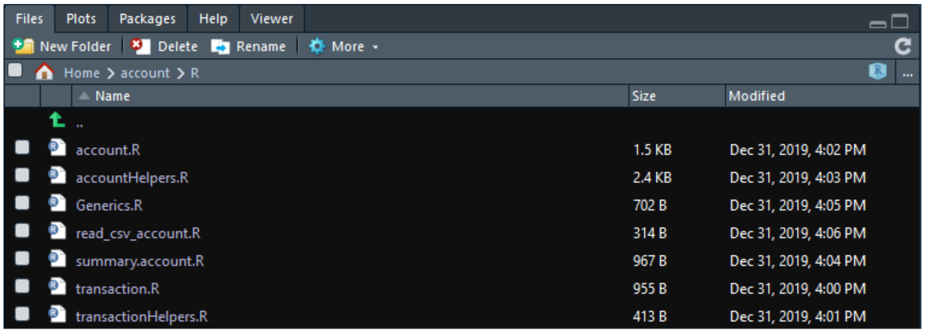
```
as.account <- function(x, ...) UseMethod("as.account")

as.account.data.frame <- function(x, title = "Account", owner = "noone",
                                  datecol = 1, amountcol = 2, memocol = 3) {
  if (length(x) < 3) stop("Too few columns to contain valid transactions")
  dat <- data.frame("date" = as.character(x[[datecol]]),
                    "amount" = as.numeric(x[[amountcol]]),
                    "memo" = as.character(x[[memocol]]),
                    stringsAsFactors = FALSE)
  acc <- Reduce(`+.account`, lapply(1:nrow(dat), function(i) {
          r <- dat[i, ]
          account_new_transaction(date = r$date, amount = r$amount,
                                  memo = r$memo)
  }))
  account_owner(acc) <- owner
  account_title(acc) <- title
  acc
}

as.account.matrix <- function(x, ...) {
  as.account(as.data.frame(x, stringsAsFactors = FALSE), ...)
}
```

`read_csv_account.R`

File containing the function `read_csv_account()`.

```
read_csv_account <- function(file, title = "Account", owner = "noone",
                             datecol = 1, amountcol = 2, memocol = 3, ...) {
  dat <- read.csv(file = file, ...)
  as.account(dat, title = title, owner = owner, datecol = datecol,
             amountcol = amountcol, memocol = memocol)
}
```

When we create these files our `R/` directory in RStudio should look like so:



Name	Size	Modified
account.R	1.5 KB	Dec 31, 2019, 4:02 PM
accountHelpers.R	2.4 KB	Dec 31, 2019, 4:03 PM
Generics.R	702 B	Dec 31, 2019, 4:05 PM
read_csv_account.R	314 B	Dec 31, 2019, 4:06 PM
summary.account.R	967 B	Dec 31, 2019, 4:04 PM
transaction.R	955 B	Dec 31, 2019, 4:00 PM
transactionHelpers.R	413 B	Dec 31, 2019, 4:01 PM

Figure 12: R directory

Now if we build the package and load it all these files are available to us.

Remember: these files should only create R objects (usually functions). These are not executable scripts that themselves run functions. That said, with **roxygen2**, you will likely be writing the documentation in these files too.

`data/`

The `data/` directory contains data sets used in the package. These could be data sets used in examples, they could be important for how certain functions work, or they could even be the reason for the existence of the package. Data sets should be stored in R data files with the `.RData` extension. The objects stored in these files can be just about anything.

Our package does have an example object, `accdata`. We will create this object ourselves. Below I've created a CSV file containing our example account.

```
date,amount,memo
1925-10-08,0.55,Initial
1925-10-14,694.81,Deposit
1925-10-14,-200.00,Withdrawal
1925-10-16,-21.25,Fee (service)
1925-10-16,-1.50,Check to General Store
1925-10-20,-2.99,Electric Bill
1925-10-21,-300.00,Withdrawal
1925-10-22,-100.00,Withdrawal
1925-10-23,-29.08,Check to General Store
1925-10-24,2.99,Deposit
1925-10-27,-6.77,Check to American Telephone & Telegraph
1925-10-28,694.81,Deposit
1925-10-30,50.00,Transfer from savings
1925-11-03,-33.55,Check to Arkham Insurance
1925-11-03,-100.00,Check to Joey Vigil
1925-11-06,-710.49,Mortgage payment
1925-11-07,-5.00,Fee (overdraft)
1925-11-08,-5.00,Fee (monthly)
1925-11-10,150.00,Transfer from savings
1925-11-14,694.81,Deposit
1925-11-14,-200.00,Withdrawal
1925-11-16,-21.25,Fee (service)
1925-11-17,-2.36,Check to General Store
1925-11-17,-40.00,Check to Ye Olde Magick Shoppe
1925-11-18,-1.22,Check to Velma's Diner
1925-11-20,-2.99,Electric Bill
1925-11-22,-200.00,Withdrawal
1925-11-23,-30.44,Check to General Store
1925-11-27,-6.81,Check to American Telephone & Telegraph
1925-11-28,694.81,Deposit
1925-11-30,-19.65,Check to General Store
1925-12-01,-150.00,Withdrawal
```

```
1925-12-03,-33.55,Check to Arkham Insurance
1925-12-06,-710.49,Mortgage payment
1925-12-08,-5.00,Fee (monthly)
```

Let's create a directory, `data-raw`, in the main directory of the package, then save this data in the file `accdata.csv`. Also, add the following line to `.Rbuildignore`:

```
^data-raw/
```

This directory simply contains raw data. We don't absolutely need it, but it's nice to have these files around in case we need them. Create directory `data/` in the main package directory. Then try executing the following (be sure the file path makes sense relative to your working directory):

```r
accdata <- read_csv_account("data-raw/accdata.csv",
                            title = "Personal Checking Accounting",
                            owner = "Joe Diamond")
save(accdata, file = "data/accdata.RData")
```

Now that the file `accdata.Rda` has our object `accdata` and has it saved in the `data/` directory, when we load a fresh R session, we should be able to use `accdata` when we load the **account** package. You can confirm this by issuing the command `rm(list = ls())` (to clear everything in the global namespace) and then in RStudio typing `Ctrl+Shift+F10` (on Windows/Linux), then rebuilding and loading the **account** package.

`man/`

The `man/` directory contains documentation for R objects. These are generally `.Rd` files, which are plain text files filled with syntax that strongly resembles LaTeX. RStudio populated this directory with a file already, `hello.Rd`, listed below:

```
\name{hello}
\alias{hello}
\title{Hello, World!}
\usage{
hello()
}
\description{
Prints 'Hello, world!'.
}
\examples{
hello()
}
```

This is the documentation for the function `hello()` that we deleted. If you were to type `?hello` in the console, you would see a rendered version of this file.

We will be using **roxygen2** for managing documentation, in which case we can mostly ignore the contents of this directory. The packages we will be using will automatically populate this directory. That said, we should delete `hello.Rd`, since we don't want documentation for a function that doesn't exist.

`vignettes/`

The `vignettes/` directory contains files for generating **vignettes**, which is long-form documentation of a package. In short, a vignette is like a tutorial, paper, or book chapter. This is where extensive tutorials would be placed, or explanations for statistical methods or algorithms used in the package. Many of the packages you work with have vignettes. Try typing `browseVignettes("roxygen2")`, `browseVignettes("devtools")`, or `browseVignettes("dplyr")` (assuming you have these packages installed). If you want to learn more about how to use a package, vignettes are a good place to start.

The documentation that we will provide with functions serves as user manuals that should explain everything a user needs to be able to use the function, explaining what each parameter does and the results. Vignettes, in comparison, contain big picture documentation. The examples provided with functions are often terse, enough to see how the function should be used. Vignettes may demonstrate package use with complete analyses.

We can choose a vignette engine that controls how vignettes are rendered. Many package authors are academics already familiar with LaTeX and so prefer writing vignettes in LaTeX. However, if you're not interested in learning LaTeX (you should, though; it's really useful), you can tell R that you want **knitr** to be the vignette engine and write your documentation in R Markdown.

Let's create a vignette introducing our package. We will make the following changes:

1. Create in the package's base directory a `vignettes/` directory.
2. In the `DESCRIPTION` file, add the following lines:

```
Suggests: knitr
VignetteBuilder: knitr
```

3. In the new `vignettes/` directory, create the file `IntroToaccount.Rmd`.

Copy and paste the following into the file `IntroToaccount.Rmd`. This will be our basic vignette.

```
---
title: "Intro to account"
author: "Curtis Miller"
date: "2020-02-14"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Intro to account}
```

~~~
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---

# Introduction

The **account** package was developed to introduce R S3 programming. It
implements tools useful for modelling banking accounts.

Try typing the following after installing the package:

```r
library(account)
acc <- account(start = "2020-01-01", title = "My Account", init = 1000,
               owner = "John Smith")
print(acc)
```

*To be continued...*
~~~

When we knit this file, the vignette should build when the package builds and we should see it when we type `browseVignettes("account")`. However I have personally had issues with vignettes apparently not being added when I build and install a package on Windows systems using `Ctrl+Shift+B`. You may need to build the package source and manually install it yourself to see that vignettes are in fact installed correctly and for `browseVignettes()` to work. See the section "Distributing Packages" section for more details.

`tests/`

The `tests/` directory contains tests that will be run when the package is checked. We will discuss test writing in a later leture, when we discuss the **testthat** package.

Other Directories

Other important directories that may appear in packages:

- `inst/` contains files that should be copied into the top-level package directory when the package is installed. I myself have used this directory to put files containing references for citations in function documentation. The `CITATION` file, containing information on how to cite the package, is put here as well.
- `src/` contains the source code for compiled code from languages such as C/C++. This is beyond the scope of this course. (I have written C++ code for my packages and I'm far from alone; this directory is definitely used.)

- `exec/` contains executable scripts. These might be needed for the package itself to function, but I have used this mostly as a place to put executable R scripts used in research projects.
- `po/` contains translations for messages. If you need your package to also be friendly to Chinese users, you may be interested in working with this directory. But it's beyond the scope of this course.
- `tools/` contains auxiliary files needed for configuration or generating scripts.
- `demo/` contains package demos, which are `.R` files that *are* executable (unlike those in `R/`). Demos are basically long examples. Vignettes and examples are better.

Things to Never Do in Packages

Writing code for packages is different from writing non-package code. If you're not going to distribute your package, you could perhaps ignore these rules, but breaking these rules would prevent the package from being accepted to CRAN, as breaking these rules is considered anti-social.

- Never load another package using `library()` or `require()` in your package. Dependencies on other packages needs to be handled via `Depends` and `Imports` in the `DESCRIPTION` file.
- Never call `q()` or `quit()` in a package. This will quit the user's session.
- Don't start external software in examples or tests unless the software is explicitly closed afterwords.
- Don't write to the user's home filespace or anywhere else in the filesystem save for the temporary directory R creates for itself when running. Don't install into the system's R installation either.
- Do not modify the global environment.

Read the CRAN repository policies to see what other behaviors might be consider "anti-social" and should not be done in packages.

Distributing Packages

When we tell RStudio to build and install a package via hotkeys, RStudio is taking the following steps:

1. It builds a package binary, the source file, a `.tar.gz` file, containing the package.
2. It installs the package from this file.

R packages are distributed via `.tar.gz` files. Users of Linux should be familiar with these files, known as **tarballs**; they're simply compressed files containing other files, similar to `.zip` files that probably most of you have used. In fact, when packages are installed via `install.packages()`, what actually happens is R downloads the corresponding tarball from CRAN and installs it, running the system commands (specifically, `R CMD install`) necessary to install the package.

Packages exist for sharing and distributing code, so we need to learn how to vproduce the installation tarball. In RStudio click on the `Build` menu, then click `Build Source Package`.

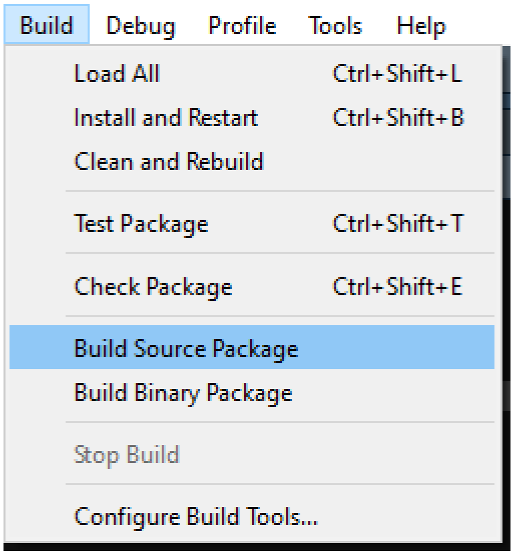


Figure 13: Build source

R will then build a tarball called `account_0.1.0.tar.gz` in the directory hosting the directory we've been using to build the package. We can in fact install the package in this file directly by clicking `Tools` in RStudio then clicking `Install Packages`.

When we do a popup window will appear. Tell R to install the package from a package archive file. This may prompt another popup window to appear; if not, click `Browse`. Select the file in the resulting file browser, or you can type the file path directly if a file browser was not used.

(This workflow installs any R package in the form of a tarball, not just ones we build.) After installation the package should be available for use.

If you're interested in submitting your package to CRAN, you will need to submit the tarball to them. But before you submit a package to CRAN, you should test the package with the CRAN checks and make sure that there were no errors or warnings in the check, and be prepared to explain any notes that are thrown as well (if they cannot be eliminated). We will not discuss submitting packages to CRAN in depth in this course.

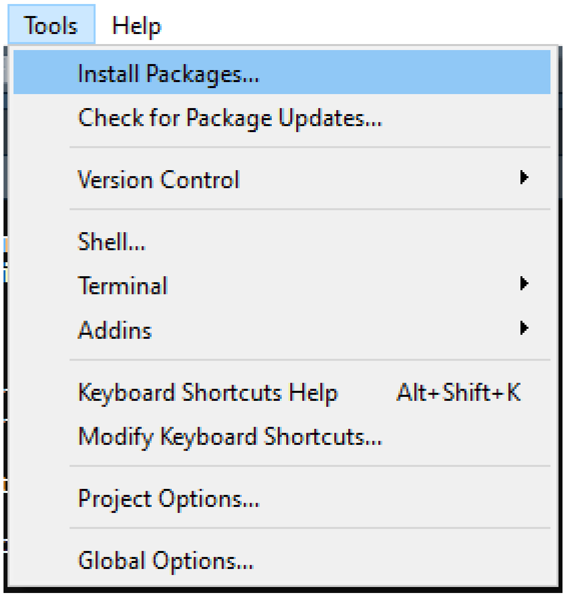
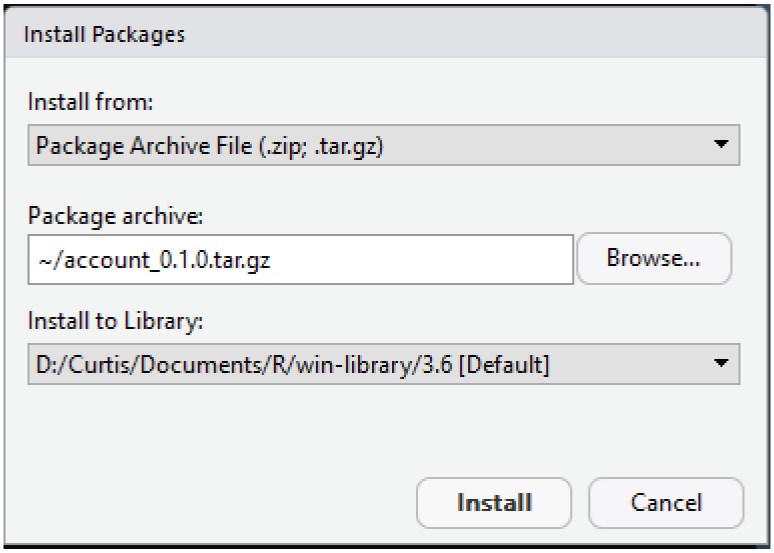


Figure 14: Install packages



Install from: Package Archive File (.zip; .tar.gz)

Package archive: ~/account_0.1.0.tar.gz

Install to Library: D:/Curtis/Documents/R/win-library/3.6 [Default]

Figure 15: Install packages popup

Lecture 6

roxygen2

roxygen2 is an R package for package development primarily doing two things: documenting objects in your package and managing `NAMESPACE`. This is done via special comments near the R code. Thus the advantages of **roxygen2** are an easier interface than writing `.Rd` files and managing `NAMESPACE` directly and keeping these important package features near the relevant code itself.

roxygen2 recognizes comments starting with `#'`. Below is an example of a function with an accompanying **roxygen2** comment:

```r
#' Add Together Two Numbers
#'
#' @param x A number
#' @param y A number
#' @return The sum of \code{x} and \code{y}
#' @examples
#' add(1, 1)
add <- function(x, y) {
  x + y
}
```

The formatting above matters. The first like is effectively the title of the function. Then `x` and `y` are listed as parameters, followed by the descriptions "A number", describing what the parameter is for. Then a description of the return value is provided. The documentation ends with an example. Everything after the `@examples` marker is executable R code. This matters; when the package is being checked, all examples will be run. If examples produce errors, an error in the package check will be thrown. If examples take too long, a warning may be thrown. The purpose of the examples are to demonstrate how the function works.

Special tags are declared using the syntax `@tag`. **roxygen2** recognizes these tags and will convert them to appropriate `.Rd` code or modify `NAMESPACE` as needed. The tags above are mostly what you need to write function documentation (there's one more useful tag, `@export`, that we will see later).

Be sure that you have **roxygen2** installed. In RStudio, reload into the workspace the **account** package (perhaps look under `File->Recent Projects`). Create a file called `add.R` in the `R/` directory, then copy and paste the above code into the file (we will be deleting this file later; it doesn't belong in our package). Then in R run the command `devtools::document()`. R runs the appropriate **roxygen2** commands, and the file `add.Rd` should appear in the `man/` subdirectory. Below are the contents of the file.

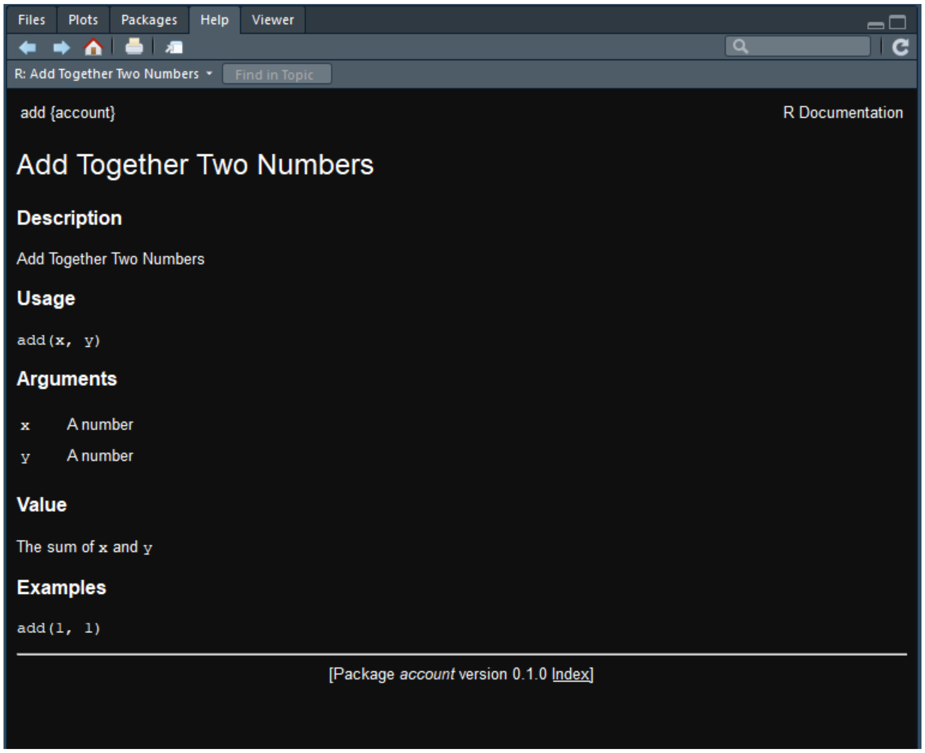
```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/add.R
\name{add}
\alias{add}
\title{Add Together Two Numbers}
\usage{
add(x, y)
}
\arguments{
\item{x}{A number}

\item{y}{A number}
}
\value{
The sum of \code{x} and \code{y}
}
\description{
Add Together Two Numbers
}
\examples{
add(1, 1)
}
```

The necessary documentation was made. Now rebuild and install **account**. When we type `?add` at the command line we get documentation for the function `add()`:

We could also type `example(add)` and the example we wrote would be run.

The result is a file resembling LaTeX but is understood by R as a documentation file. **roxygen2** ultimately produces these files and so when writing function documentation these files' conventions are respected. In particular, we may need to invoke commands that mark text as bold, italic, as code, or as itemized/enumerated lists, and so on. Calls to these commands resembles the format `\command{input}`, `\command{input1}{input2}`, or `\command[param]{input}`.

See the corresponding chapter of *R Packages* for a brief list of some useful commands (along with a more detailed discussion of using **roxygen2**, such as other tags to use). Below are commands I use frequently:



add {account} R Documentation

Add Together Two Numbers

Description

Add Together Two Numbers

Usage

`add(x, y)`

Arguments

`x`	A number
`y`	A number

Value

The sum of `x` and `y`

Examples

`add(1, 1)`

[Package *account* version 0.1.0 Index]

Figure 16: Resulting formatted documentation

- `\code{}` for formatting text to look like `code`
- `\emph{}` for *italics*
- `\strong{}` for **bold**
- `\link{}` for linking to other documentation pages, either internal or external to the package. `\code{\link{function}}` provides a link to a page referenced by `function` (probably the function's name) while formatting the text to look like code, while `\code{\link[package]{function}}` does the same except that `function` is in the package `package`.
- `\eqn{}` allows for LaTeX-style mathematical formulas and equations. `\deqn{}` is similar except that the equation is printed in display mode rather than inline mode.
- `\dontrun{}` is used to mark code in examples as code not to be run, such as `\dontrun{sum("An error!")}`. This could be because the code would produce errors (this would result in errors during package checks) or because the code would take a very long time to run.
- `\insertRef{}{}` is a macro provided by the package **Rdpack** for citation and reference management. See the **Rdpack** vignette "Inserting references in Rd and roxygen2 documentation" for more information. I'm not going to discuss this further other than to make a point: documentation needs to be handled like academic work. This means that if you're implementing a published procedure or algorithm, or even just refer to a published work in the documentation, you should cite the publication. Failure to do so is **plagiarism**. Remember: citations are not just to make sure people are appropriately credited. They also give your users something else to refer to in order to learn more about the methods available in the package. (In this course, don't worry about citations, but in the real world, always cite.) On the flip side, if you use a package in your own papers or research, including R packages (or even just R itself), you should cite the package you use. The function `citation("packagename")` will even print out the citation information.

`NAMESPACE` Management

When a function or object should be made available for users to use, add the line `#' @export` to the documentation. **roxygen2** is also for managing the `NAMESPACE` file, and will update the file so that objects that should be made available to the user are in fact available. However, by default objects are *not* available (at least not without using `:::`), so we need to decleare objects accessable to the user via `@export`. Any objects not exported are visable internally to the package and can be used in package code without any special syntax, but will not be available to users. Thus use `@export` only for the code that is necessary to use the package by users.

There is nothing that *must* be done to make an object internal only, but adding `#' @keywords internal` can be useful; this tag would notify **roxygen2** and the R package builder that this function is internal, so while documentation for the function should still be made, it should not be included in the general documentation index or the PDF manual accompanying the package, and the examples for the function should not be run in the automated tests. The documentation still exists, though, and users of the package coming from a development angle can still read it; since most users won't be interested, though, the documentation should not be prominent.

Right now the `NAMESPACE` file for **account** has a single line that effectively means "export everything named". We don't want this. Delete `NAMESPACE`. Then add the line `#' @export` right before the line `#' examples` in `R/add.R`. Execute `devtools::document()` in the console. In addition to documentation files being rebuilt, a new `NAMESPACE` file is created, with the following lines:

```
# Generated by roxygen2: do not edit by hand

export(add)
```

The `add()` function is now visible to package users. In fact, it's the *only* function available to users; all the others are invisible and can only be accessed via `account:::`. We will determine which functions to export later.

You may delete `R/add.R` and `man/add.Rd`; `add()` has nothing to do with **account**'s main functionality and should be removed.

Documenting Functions

See the above discussion for documenting R functions; there's not much more to add, at least for these lecture notes. That said, functions that modify in place (such as `names<-()`) likely don't need special documentation; document the original function `names()` and mention the ability to modify in place, with examples.

Documenting S3 Classes

S3 classes are so loosely defined there's no special documentation for them. Hopefully you write a constructor function for these objects; document the constructor like you would any function, but be sure there's adequate discussion of the structure of what's returned.

S3 generic functions are also just functions and can be documented like any other function. Generic function methods, again, are just like any other function. However, you may want to consider adding a "See also" section via the tag `@seealso` and refer to the function methods so that users can see how the generic function should be used.

Should you `@export` objects meant to work with S3 classes, such as constructors, generics, and methods? If you want users to be able to create instances of a class using the constructor, you should export it. In fact, you're less likely to introduce bugs if you default to exporting every method you write, even methods for generic functions that are not exported. If you have a method for a generic in an imported package, you should import the generic from the foreign package then export the method.

Documenting Data Sets

For data sets tags such as `@param` and `@return` are meaningless. I generally use two tags for documenting data sets: `@format` and `@source`. `@format` should include a description of how the data set is structured. For example with a data frame, you should describe how many rows and columns are along with describing what data is in each column, what type of data it is, and other important information (such as units, how the data was collected, etc.). `@source` may be just a URL to an internet source the data was obtained from (wrapped in `\url{}`), or perhaps a paper citation if the data set was published. Just describe where the data came from. (Perhaps also consider providing `@examples` and include demo analyses of the data set.)

The data set lives in a `.RData` file in the `data/` directory so we can't put the comments right above it. Instead, we would place the comments right before a character string naming the data set. For example:

```
#' Data Set
#'
#' An awesome data set
#'
#' @format Univariate \code{numeric} vector
#'
#' @source \url{http://www.some.url/}
"awesome_data"
```

Package-Level Documentation

Often packages come with an overview documentation file. There is no object that goes with a package; we work around this by documenting `NULL`. Thus we can have a documentation file for the package that looks like so:

```
#' foo: A package for bar
#'
#' The foo package provides bar and baz.
#'
#' @docType package
#' @name foo
NULL
```

The `@docType` tag manually marks this documentation as package-level documentation , and the `@name` tag names the documentation file (normally this would automatically be detected for functions).

When managing `NAMESPACE`, the package-level documentation comments make for a great place to place the tag `@import` to import other packages. We can import the package **dplyr** via a comment line `#' @import dplyr`. If we use **dplyr** a lot in our package we should do this. Additionally, if there is a specific function we wish to import, we should add a line resembling `#' importFrom package function`.

account Example Documentation

Let's demonstrate documentation by properly documenting the objects in the **account** package. I'm also going to give each `.R` file some header and organizational comments. I personally like header comments as they immediately give code readers a description of what's in the file. Section comments demarcate sections of related code.

I use the following format:

```
################################################################################
# FileName.R
# Author
# Date
################################################################################
# This is a brief description of the file
################################################################################

################################################################################
# SECTION
################################################################################
```

You can take the following code listings and directly copy and paste them into the corresponding files in `R/`. Note that there is a new `.R` file, `Data.R`, documenting both data sets and the overall package.

`transaction.R`

This file includes `transaction()`, the function for making `transaction`-class objects, along with associated methods.

```
################################################################################
# transaction.R
# Curtis Miller
# 2020-01-06
################################################################################
# Defines the transaction class constructor with associated methods.
################################################################################

################################################################################
# CONSTRUCTOR
```

```
################################################################################

#' Bank Account Transactions Class Constructor
#'
#' This is the constructor function for \code{transaction}-class objects, which
#' represent financial transactions in an account.
#'
#' @param date Any input that can be understood by \code{\link[base]{as.Date}}
#'             to represent a date
#' @param amount Numeric representing the transaction amount; values below zero
#'               are withdrawals while values above zero are deposits
#' @param memo Optional memo field
#' @return A \code{transaction}-class object, which is a
#'         \code{\link[base]{list}} with the following entries:
#'         \describe{
#'           \item{\code{date}}{A \code{Date}-class object representing the date
#'                              of the transaction}
#'           \item{\code{amount}}{The amount of the transaction (a numeric)}
#'           \item{\code{memo}}{A character string, an associated note with the
#'                              transaction}
#'         }
#' @export
#' @examples
#' transaction("2010-01-01", 1000, "Initial")
#' transaction("2010-01-02", -20)
transaction <- function(date, amount, memo = "") {
  obj <- list(date = as.Date(date),
              amount = as.numeric(amount),
              memo = as.character(memo)
  )
  class(obj) <- "transaction"
  obj
}

################################################################################
# HELPERS
################################################################################

#' Detect \code{transaction}-class Objects
#'
#' Detects whether an input object is a \code{\link{transaction}}-class object.
#'
#' @param x An input object
#' @return A logical value, \code{TRUE} if \code{x} is a
#'         \code{\link{transaction}}-class object, \code{FALSE} otherwise
#' @export
#' @examples
#' is.transaction("not a transaction")
#' is.transaction(transaction("2010-01-01", 11))
is.transaction <- function(x) {class(x) == "transaction"}

################################################################################
# METHODS
################################################################################

#' Print \code{transaction}-class Objects
#'
#' Print \code{\link{transaction}}-class objects, recognizing their structure.
#'
#' @param x A \code{\link{transaction}}-class object to print.
#' @param space Specify the space preceding the \code{Date} field
#' @export
#' @examples
#' print(transaction("2010-01-01", 1000, "Initial"))
print.transaction <- function(x, space = 10) {
  tdate <- as.character(x$date)
  datestring <- paste0("    ", tdate, ":")
  formatstring <- paste0("%+", space[[1]], ".2f")  # See sprintf() to explain
  amountstring <- sprintf(formatstring, x$amount)
  if (x$memo == "") {
    memostring <- ""
  } else {
    memostring <- paste0("(", x$memo, ")")
  }
  cat(datestring, amountstring, memostring, "\n")
}

#' Form an \code{account}-class Object from a \code{transaction}-class Object
#'
#' Using an input \code{\link{transaction}}-class object, construct a
#' \code{\link{account}}-class object.
#'
#' @param x The \code{\link{transaction}}-class object to convert
#' @param title A character string representing the resulting
#'              \code{\link{account}}-class object's title
#' @param owner A character string representing the resulting
#'              \code{\link{account}}-class object's owner
#' @return An \code{\link{account}}-class object with the input
#'            \code{\link{transaction}} \code{x} as the only transaction in the
#'            account (\strong{beware:} most functions working with
#'            \code{\link{account}}-class objects expect the object to have a
#'            transaction with the memo field \code{"Initial"}, which this
#'            function does not guarantee)
#' @seealso \code{\link{as.account}}, \code{\link{account}}
#' @export
#' @examples
#' u <- transaction("2010-01-01", 1000, "Initial")
#' as.account(u, title = "My Account", owner = "John Doe")
as.account.transaction <- function(x, title = "Transaction", owner = "noone") {
  res <- account(start = transaction_date(x), init = amount(x), owner = owner,
                 title = title)
  memo(account_transactions(res)[[1]]) <- memo(x)
  res
}
```

`transactionHelpers.R`

This file includes functions that are meant to work with `transaction`-class objects. Here I needed to use the `@rdname` tag to put the documentation of multiple functions in the same file. This tag, along with the similar tag `@describeIn`, allows the documentation of similar functions to be put in the same `.Rd` file, thus potentially allowing for cleaner documentation. (Use sparingly, though, when the functions are highly similar.) Common fields are appended onto the original object's documentation.

```
################################################################################
# transactionHelpers.R
# Curtis Miller
# 2020-01-06
################################################################################
# Helper functions for working with transaction-class objects
################################################################################

################################################################################
# DATES
################################################################################

#' Get and Manipulate \code{transaction}-Class Object Dates
#'
#' Functions to get or set dates associated with \code{\link{transaction}}-class
#' objects.
#'
#' @param trns The \code{\link{transaction}}-class object
#' @return If anything, the date of the transaction
#' @export
#' @seealso \code{\link{transaction}}
#' @examples
#' u <- transaction("2010-01-01", 1000, "Initial")
#' transaction_date(u)
transaction_date <- function(trns) {
  trns$date
}

#' @param value Any input that can be understood by \code{\link[base]{as.Date}}
#'              to mean a date
#' @export
#' @rdname transaction_date
#' @examples
#' transaction_date(u) <- "2010-01-02"
#' print(u)
`transaction_date<-` <- function(trns, value) {
  trns$date <- as.Date(value)
  trns
}

################################################################################
# AMOUNTS
################################################################################

#' Get and Manipulate \code{transaction}-Class Object Amounts
#'
#' Functions to get or set amounts associated with \code{\link{transaction}}-class
#' objects.
#'
#' @param trns The \code{\link{transaction}}-class object
#' @return If anything, the amount of the transaction
#' @export
#' @seealso \code{\link{transaction}}
#' @examples
#' u <- transaction("2010-01-01", 1000, "Initial")
#' amount(u)
amount <- function(trns) {
  trns$amount
}

#' @param value Numeric for the new amount of the transaction
#' @export
#' @rdname amount
#' @examples
#' amount(u) <- 2000
#' print(u)
```

```r
`amount<-` <- function(trns, value) {
  trns$amount <- as.numeric(value)
  trns
}

################################################################################
# MEMOS
################################################################################

#' Get and Manipulate \code{transaction}-Class Object Memos
#'
#' Functions to get or set memos associated with \code{\link{transaction}}-class
#' objects.
#'
#' @param trns The \code{\link{transaction}}-class object
#' @return If anything, the memo of the transaction
#' @export
#' @seealso \code{\link{transaction}}
#' @examples
#' u <- transaction("2010-01-01", 1000, "Initial")
#' memo(u)
memo <- function(trns) {
  trns$memo
}

#' @param value The new memo of the transaction
#' @export
#' @rdname memo
#' @examples
#' memo(u) <- "Nothing"
#' print(u)
`memo<-` <- function(trns, value) {
  trns$memo <- as.character(value)
  trns
}
```

account.R

This file includes `account()`, the function for making `account`-class objects, along with associated methods.

```r
################################################################################
# account.R
# Curtis Miller
# 2020-01-06
################################################################################
```

```
# Defines a constructor and methods for account-class objects.
################################################################################

################################################################################
# CONSTRUCTOR
################################################################################

#' Financial Account Class Constructor
#'
#' This is the constructor function for \code{account}-class objects, which
#' represent financial accounts with a sequence of transactions.
#'
#' @param start Any object that \code{\link[base]{as.Date}} can understand as a
#'              date, representing the start date of the account
#' @param owner A character string representing the owner of the account
#' @param init The initial amount of money in the account
#' @param title A character string titling the account
#' @return An \code{account}-class object, which is a \code{\link[base]{list}}
#'         with the following entries:
#'         \describe{
#'           \item{\code{title}}{A character string representing the account
#'                               title}
#'           \item{\code{owner}}{A character string representing the owner of
#'                               the account}
#'           \item{\code{transactions}}{A \code{\link[base]{list}} of
#'                                      \code{\link{transaction}}-class objects
#'                                      representing the account's transactions}
#'         }
#' @seealso \code{\link{as.account}}
#' @export
#' @examples
#' (acc <- account(start = "2010-01-01", owner = "John Doe",
#'                 title = "My Account", init = 1000))
#' acc <- acc + as.account(transaction("2010-01-02", -300, "Withdrawal"))
#' acc <- acc + as.account(transaction("2010-01-03", -500, "Withdrawal"))
#' acc <- acc + as.account(transaction("2010-01-04", 500, "Deposit"))
#' acc <- acc + as.account(transaction("2010-01-04", -20, "Check"))
#' summary(acc)
account <- function(start, owner, init = 0, title = "Account") {
  obj <- list(
    title = as.character(title),
    owner = as.character(owner),
    transactions = list(transaction(start, init, "Initial"))
  )
  class(obj) <- "account"
  obj
}

################################################################################
# HELPERS
################################################################################

#' Detect \code{account}-class Objects
#'
#' Detects whether an input object is a \code{\link{account}}-class object.
#'
#' @param x An input object
#' @return A logical value, \code{TRUE} if \code{x} is a
#'         \code{\link{account}}-class object, \code{FALSE} otherwise
#' @export
#' @examples
#' is.account("not an account")
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' is.account(acc)
is.account <- function(x) {class(x) == "account"}

################################################################################
# METHODS
################################################################################

#' Sort \code{account}-class Object Transactions by Date
#'
#' Sorts an \code{\link{account}}-class objects so that the transactions in the
#' account have ordered dates, with the initial transaction being placed first.
#'
#' @param x The \code{\link{account}}-class object to sort
#' @param decreasing Logical; if \code{TRUE}, then transactions are sorted in
#'                   descending order (most recent transactions first)
#' @param ... Other arguments to pass to \code{\link[base]{order}}
#' @return The sorted \code{\link{account}} object
#' @export
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' acc <- acc + as.account(transaction("2010-01-02", -300, "Withdrawal"))
#' acc <- acc + as.account(transaction("2010-01-03", -500, "Withdrawal"))
#' acc <- acc + as.account(transaction("2010-01-04", 500, "Deposit"))
#' acc <- acc + as.account(transaction("2010-01-04", -20, "Check"))
#' sort(acc, decreasing = TRUE)
```

```r
sort.account <- function(x, decreasing = FALSE, ...) {
  # There might be multiple entries with memo Initial; design code for that
  memo_Initial <- which(account_memos(x) == "Initial")
  if (length(memo_Initial) == 0) {
    date_Initial <- min(account_dates(x))
    true_Initial <- which.min(account_dates(x))
  } else {
    date_Initial <- account_dates(x)[memo_Initial]
    true_Initial <- which((account_dates(x) == min(date_Initial)) &
                          (account_memos(x) == "Initial"))
  }
  tcount <- transaction_count(x)
  nix <- (1:tcount)[-true_Initial]
  ordered_nix <- nix[order(account_dates(x)[nix], decreasing = decreasing, ...)]
  if (decreasing) {
    final_order <- c(ordered_nix, true_Initial)
  } else {
    final_order <- c(true_Initial, ordered_nix)
  }
  account_transactions(x) <- account_transactions(x)[final_order]
  x
}

#' Print \code{account}-class Objects
#'
#' Print \code{\link{account}}-class objects, recognizing their structure.
#'
#' @param x A \code{\link{account}}-class object
#' @param presort Logical; if \code{TRUE}, the transactions in \code{x} are
#'                sorted before printing
#' @seealso \code{\link{print.transaction}}
#' @export
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' acc <- acc + as.account(transaction("2010-01-02", -300, "Withdrawal"))
#' acc <- acc + as.account(transaction("2010-01-03", -500, "Withdrawal"))
#' acc <- acc + as.account(transaction("2010-01-04", 500, "Deposit"))
#' acc <- acc + as.account(transaction("2010-01-04", -20, "Check"))
#' print(acc)
print.account <- function(x, presort = FALSE) {
  if (presort) {
    x <- sort(x)
  }
  cat("Title:", account_title(x))
  cat("\nOwner:", account_owner(x))
  cat("\nTransactions:\n-----------------------------------------------------\n")
  for (t in account_transactions(x)) {
    print(t)
  }
}

#' Addition of Accounts
#'
#' Overloading of the \code{+} operator to allow for addition between two
#' \code{\link{account}}-class objects
#'
#' @param x An \code{\link{account}}-class object
#' @param y See \code{x}
#' @return An \code{\link{account}}-class object resembling \code{x} except
#'         including all transactions in \code{y} and sorted transactions
#' @export
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' acc <- acc + as.account(transaction("2010-01-02", -300, "Withdrawal"))
#' acc <- acc + as.account(transaction("2010-01-03", -500, "Withdrawal"))
#' acc <- acc + as.account(transaction("2010-01-04", 500, "Deposit"))
#' acc <- acc + as.account(transaction("2010-01-04", -20, "Check"))
#' print(acc)
`+.account` <- function(x, y) {
  account_transactions(x) <- c(account_transactions(x), account_transactions(y))
  sort(x)
}

#' Plot Account Balance
#'
#' Plot the balance of an \code{\link{account}}-class object.
#'
#' @param x The \code{\link{account}}-class object
#' @param y Not used
#' @param ... Additional arguments to pass to \code{\link[base]{plot}}
#' @return The result of \code{\link[base]{plot}}
#' @export
#' @examples
#' plot(accdata)
plot.account <- function(x, y, ...) {
  if (bad_Initial(x)) stop("Malformed account object")
  x <- sort(x)
  unique_dates <- unique(account_dates(x))
```

```r
  date_trans_sum <- sapply(unique_dates, function(d) {
    idx <- which(account_dates(x) == d)
    sum(account_trans_amounts(x)[idx])
  })
  plot(unique_dates, cumsum(date_trans_sum), type = "l",
       main = account_title(x), xlab = "Date", ylab = "Balance", ...)
}
```

`accountHelpers.R`

This file includes functions that are meant to work with `account`-class objects. Note that some functions here are not exported to the global namespace, as they are meant for internal use only. Additionally, the tag `@inheritParams` is used to copy the documentation of the parameters of one function directly to another function. Specifically, `#' @inheritParams foo` when used in the documentation for function `bar()` will cause the documentation of the parameters of `foo()` that isn't written (overridden) in the documentation of `bar()` to appear in the documentation of `bar()`. This can be useful when two functions share parameters, or one function inherits the parameters of another.

```r
################################################################################
# accountHelpers.R
# Curtis Miller
# 2020-01-06
################################################################################
# Helper functions for working with account-class objects
################################################################################

################################################################################
# ACCOUNT FIELDS
################################################################################

#' Get or Set Account Title
#'
#' Get or set the title of an \code{\link{account}}-class object.
#'
#' @param account The \code{\link{account}}-class object
#' @return The title of \code{\link{account}}
#' @export
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' account_title(acc)
account_title <- function(account) {
  account$title
}
```

```
#' @param value The new title of the transaction
#' @export
#' @rdname account_title
#' @examples
#' account_title(acc) <- "Nothing"
#' print(acc)
`account_title<-` <- function(account, value) {
  account$title <- as.character(value)
  account
}

#' Get or Set Account Owner
#'
#' Get or set the owner of an \code{\link{account}}-class object.
#'
#' @param account The \code{\link{account}}-class object
#' @return The owner of \code{\link{account}}
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                 title = "My Account", init = 1000)
#' account_owner(acc)
account_owner <- function(account) {
  account$owner
}

#' @param value The new owner of the transaction
#' @export
#' @rdname account_owner
#' @examples
#' account_owner(acc) <- "noone"
#' print(acc)
`account_owner<-` <- function(account, value) {
  account$owner <- as.character(value)
  account
}

#' Get or Set Account Transactions
#'
#' Get or set the list of transactions associated with an
#' \code{\link{account}}-class object.
#'
#' Please avoid using this function for adding transactions to an
#' \code{\link{account}}-class object; other functions (such as overloaded
#' \code{+}) should be used instead as they provide a safer interface.
#'
```

```r
#' @param account The \code{\link{account}}-class object
#' @return The \code{\link[base]{list}} of the \code{\link{account}} object's
#'         \code{\link{transaction}}s.
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' account_transactions(acc)
account_transactions <- function(account) {
  account$transactions
}

#' @param value The new list of transactions
#' @export
#' @rdname account_transactions
`account_transactions<-` <- function(account, value) {
  account$transactions <- value
  account
}

################################################################################
# ACCOUNT COLLECTIVE PROPERTIES
################################################################################

#' Determine if Account's Transactions are \code{transaction}-class
#'
#' Check if an \code{\link{account}}-class object has appropriate transactions
#'
#' @param account The \code{\link{account}}-class object to check
#' @return If all objects in \code{account}'s transaction list are
#'         \code{\link{transaction}}-class objects, then this returns
#'         \code{TRUE}; otherwise, it returns \code{FALSE}
#' @keywords internal
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' account:::all_transactions(acc)
all_transactions <- function(account) {
  all(sapply(account_transactions(account), is.transaction))
}

#' Get Dates of All Transactions in Account
#'
#' Extract the dates of all transactions in an \code{\link{account}}-class
#' object and return a vector containing the dates. (Note: dates not sorted.)
#'
```

```
#' @param account The \code{\link{account}}-class object
#' @return A vector of \code{Date}s associated with transactions
#' @seealso \code{\link{transaction}}, \code{\link{transaction_date}}
#' @export
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' acc <- acc + account_new_transaction("2010-01-02", -300, "Withdrawal")
#' acc <- acc + account_new_transaction("2010-01-03", -500, "Withdrawal"))
#' acc <- acc + account_new_transaction("2010-01-04", 500, "Deposit"))
#' acc <- acc + account_new_transaction("2010-01-04", -20, "Check")
#' account_dates(acc)
account_dates <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  Reduce(c, lapply(account_transactions(account), transaction_date))
}

#' Get Amounts of All Transactions in Account
#'
#' Extract the amount of all transactions in an \code{\link{account}}-class
#' object and return a vector containing the amounts. (Note: not sorted.)
#'
#' @param account The \code{\link{account}}-class object
#' @return A numeric vector containing transaction amounts
#' @seealso \code{\link{transaction}}, \code{\link{amount}}
#' @export
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' acc <- acc + account_new_transaction("2010-01-02", -300, "Withdrawal")
#' acc <- acc + account_new_transaction("2010-01-03", -500, "Withdrawal"))
#' acc <- acc + account_new_transaction("2010-01-04", 500, "Deposit"))
#' acc <- acc + account_new_transaction("2010-01-04", -20, "Check")
#' account_trans_amounts(acc)
account_trans_amounts <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  sapply(account_transactions(account), amount)
}

#' Get Memos of All Transactions in Account
#'
#' Extract the memo of all transactions in an \code{\link{account}}-class
#' object and return a vector containing the memo. (Note: not sorted.)
#'
#' @param account The \code{\link{account}}-class object
#' @return A character vector containing transaction memos
#' @seealso \code{\link{transaction}}, \code{\link{memo}}
#' @export
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' acc <- acc + account_new_transaction("2010-01-02", -300, "Withdrawal")
#' acc <- acc + account_new_transaction("2010-01-03", -500, "Withdrawal"))
#' acc <- acc + account_new_transaction("2010-01-04", 500, "Deposit"))
#' acc <- acc + account_new_transaction("2010-01-04", -20, "Check")
#' account_memos(acc)
account_memos <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  sapply(account_transactions(account), memo)
}

################################################################################
# ACCOUNT STATISTICS
################################################################################

#' Account Transaction Count
#'
#' Count the number of transactions in an \code{\link{account}}-class object
#'
#' @param account The \code{\link{account}}-class object
#' @return Integer representing the number of transactions in \code{account}
#' @export
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' acc <- acc + account_new_transaction("2010-01-02", -300, "Withdrawal")
#' acc <- acc + account_new_transaction("2010-01-03", -500, "Withdrawal"))
#' acc <- acc + account_new_transaction("2010-01-04", 500, "Deposit"))
#' acc <- acc + account_new_transaction("2010-01-04", -20, "Check")
#' transaction_count(acc)
transaction_count <- function(account) {
  length(account_transactions(account))
}

#' Check For Bad \code{account} Initial Value
#'
#' Check that an \code{\link{account}}-class account has an initial transaction
#' (a transaction with the \code{"Initia"} memo) that is the earliest
#' transaction in the account.
#'
```

```
#' @param account The \code{\link{account}}-class object
#' @return \code{TRUE} if there is a transaction with the assumed properties,
#'         \code{FALSE} otherwise
#' @keywords internal
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' account:::bad_Initial(acc)
#' bcc <- as.account(transaction("2010-01-02", 100, "Nothing"))
#' account:::bad_Initial(bcc)
bad_Initial <- function(account) {
  if (!all_transactions(account)) stop("Not all transactions of right class")
  if (!("Initial" %in% account_memos(account))) stop("No Initial transaction")

  memo_Initial <- which(account_memos(account) == "Initial")
  date_Initial <- min(account_dates(account)[memo_Initial])
  sort(account_dates(account))[1] < date_Initial
}

################################################################################
# MODIFICATION TOOLS
################################################################################

#' New Transaction to Add to Account
#'
#' Create an \code{\link{account}}-class object with a single transaction that
#' can then be added to another \code{\link{account}}-class object via the
#' overloaded operator \code{+}.
#'
#' @inheritParams transaction
#' @return An \code{\link{account}}-class object (with title
#'         \code{"Transaction"} and owner \code{"noone"}) with a single
#'         transaction, based on the passed parameters
#' @seealso \code{\link{account}}, \code{\link{transaction}}
#' @export
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' acc <- acc + account_new_transaction("2010-01-02", -300, "Withdrawal")
#' acc <- acc + account_new_transaction("2010-01-03", -500, "Withdrawal"))
#' acc <- acc + account_new_transaction("2010-01-04", 500, "Deposit"))
#' acc <- acc + account_new_transaction("2010-01-04", -20, "Check")
#' print(acc)
account_new_transaction <- function(...) {
  as.account(transaction(...))
```

```
}

#' Delete a Transaction from an Account
#'
#' Delete a transaction from an \code{\link{account}}-class object based on
#' matching specified lists of dates and memo fields. (At least one must be
#' specified.)
#'
#' @param account The \code{\link{account}}-class object
#' @param date An object convertible to a \code{Date} by
#'             \code{\link[base]{as.Date}} representing dates to delete
#' @param memo A character vector containing memo fields based on which
#'             matching transactions should be removed
#' @return An \code{\link{account}}-class object with transactions having
#'         (simultaneously) matching \code{date} and \code{memo} fields removed
#' @seealso \code{\link{transaction}}
#' @export
#' @examples
#' acc <- account(start = "2010-01-01", owner = "John Doe",
#'                title = "My Account", init = 1000)
#' acc <- acc + account_new_transaction("2010-01-02", -300, "Withdrawal")
#' acc <- acc + account_new_transaction("2010-01-03", -500, "Withdrawal"))
#' acc <- acc + account_new_transaction("2010-01-04", 500, "Deposit"))
#' acc <- acc + account_new_transaction("2010-01-04", -20, "Check")
#' print(acc)
#' acc <- account_delete_transaction(acc, date = "2010-01-04", memo = "Check")
#' acc <- account_delete_transaction(acc, memo = "Withdrawal")
#' print(acc)
account_delete_transaction <- function(account, date = NULL, memo = NULL) {
  if (is.null(date) && is.null(memo)) {
    stop("Must specify at least one of date or memo")
  }

  if (!is.null(date)) {
    filter_dates <- which(account_dates(account) == as.Date(date))
  } else {
    filter_dates <- 1:transaction_count(account)
  }

  if (!is.null(memo)) {
    filter_memos <- which(account_memos(account) == memo)
  } else {
    filter_memos <- 1:transaction_count(account)
  }
  final_filter <- intersect(filter_dates, filter_memos)
  if (length(final_filter) == 0) {
    warning("No transactions with date/memo combination")
    return(account)
  } else {
    account_transactions(account)[final_filter] <- NULL
    return(account)
  }
}
```

summary.account.R

This file includes `summary.account()`, the function for making `summary.account`-class objects, along with associated methods.

```
################################################################################
# summary.account.R
# Curtis Miller
# 2020-01-06
################################################################################
# Summaries of account-class object, via a method that also is a constructor of
# account.summary-class objects, with associated methods
################################################################################

################################################################################
# CONSTRUCTOR
################################################################################

#' Account Summary
#'
#' Provide a summary of an \code{\link{account}}-class object, including title,
#' owner, balance, number of transactions, and recent transactions.
#'
#' @param object The \code{\link{account}}-class object
#' @param recent The number of recent transactions to include
#' @return A \code{summary.account}-class object, which is a list with the
#'         following elements:
#'         \describe{
#'           \item{\code{title}}{The title of the account}
#'           \item{\code{owner}}{The owner of the account}
#'           \item{\code{balance}}{The balance of the account (sum of
#'                                 transactions)}
#'           \item{\code{tcount}}{Integer representing the number of
#'                                transactions in the account}
#'           \item{\code{rtrans}}{A list of recent transactions}
#'         }
#' @export
#' @examples
#' summary(accdata)
summary.account <- function(object, recent = 5) {
  if (bad_Initial(object)) warning("account object malformed!")
  res <- list()
  res$title <- account_title(object)
  res$owner <- account_owner(object)
  res$balance <- sum(account_trans_amounts(object))
  res$tcount <- transaction_count(object)
  res$rtrans <- account_transactions(sort(object,
                                         decreasing = TRUE)
                                    )[1:min(recent, res$tcount)]

  class(res) <- "summary.account"
  res
}

################################################################################
# METHODS
################################################################################

#' Print Summaries of Accounts
#'
#' Print the results of \code{summary(acc)} when \code{acc} is an
#' \code{\link{account}}-class object.
#'
#' @param x The \code{\link{account}}-class object
#' @param prefix The prefix character for the title of the summary
#' @export
#' @examples
#' print(summary(accdata))
print.summary.account <- function(x, prefix = "\t") {
  cat("\n")
  cat(strwrap(x$title, prefix = prefix), sep = "\n")
  cat("\n")
  cat("Owner:  ", sprintf("%20s", x$owner), "\n", sep = "")
  cat("Transactions:  ", sprintf("%13s", x$tcount), "\n", sep = "")
  cat("Balance:  ", sprintf("%18.2f", x$balance), "\n", sep = "")
  cat("\nRecent Transactions:\n----------------------------------------------------\n")
  for (t in x$rtrans) {
    print(t)
  }
}
```

Generics.R

This file defines generic functions, along with any methods for external classes.

```
################################################################################
# Generics.R
# Curtis Miller
# 2020-01-06
################################################################################
# Generic function declaration, with accompanying methods for external classes.
################################################################################

################################################################################
# GENERICS
################################################################################

#' Convert Objects to \code{account}-class Objects
#'
#' Convert objects to \code{\link{account}}-class objects.
#'
#' @param x The object to convert to an \code{\link{account}}-class object
#' @param ... Additional parameters for conversion
#' @return An \code{\link{account}}-class object
#' @seealso \code{\link{account}}, \code{\link{as.account.transaction}},
#'          \code{\link{as.account.data.frame}}
#' @export
#' @examples
#' u <- transaction("2010-01-01", 100, "Initial")
#' as.account(u)
#' dframe <- data.frame(date = c("2010-01-01", "2010-01-02", "2010-01-03"),
#'                      amount = c(100, -20, -50),
#'                      memo = c("Initial", "Withdrawal", "Check"))
#' as.account(dframe)
as.account <- function(x, ...) UseMethod("as.account")

################################################################################
# METHODS
################################################################################

#' Convert Data Frame Data to an \code{account}-class Object
#'
#' Convert the information in a \code{\link[base]{data.frame}} to transactions
#' of an \code{\link{account}}-class object.
#'
#' @param x The \code{\link[base]{data.frame}} to convert
#' @param datecol An identifier for the column of \code{x} representing
#'                 transaction dates
#' @param amountcol An identifier for the column of \code{x} representing
#'                   transaction amounts
#' @param memocol An identifier of the column of \code{x} representing
#'                 transaction memos
#' @inheritParams account
#' @return An \code{\link{account}}-class object
#' @seealso \code{\link{as.account}}, \code{\link{account}}
#' @export
#' @examples
#' dframe <- data.frame(date = c("2010-01-01", "2010-01-02", "2010-01-03"),
#'                       amount = c(100, -20, -50),
#'                       memo = c("Initial", "Withdrawal", "Check"))
#' as.account(dframe, title = "My Account", owner = "John Doe")
as.account.data.frame <- function(x, title = "Account", owner = "noone",
                                  datecol = 1, amountcol = 2, memocol = 3) {
  if (length(x) < 3) stop("Too few columns to contain valid transactions")
  dat <- data.frame("date" = as.character(x[[datecol]]),
                    "amount" = as.numeric(x[[amountcol]]),
                    "memo" = as.character(x[[memocol]]),
                    stringsAsFactors = FALSE)
  acc <- Reduce(`+.account`, lapply(1:nrow(dat), function(i) {
          r <- dat[i, ]
          account_new_transaction(date = r$date, amount = r$amount,
                                  memo = r$memo)
  }))
  account_owner(acc) <- owner
  account_title(acc) <- title
  acc
}

#' Convert Matrix Data to an \code{account}-class Object
#'
#' Convert data stored in a \code{\link[base]{matrix}} into transactions of an
#' \code{\link{account}}-class object. This function will likely fail if not
#' given a matrix of character data. The matrix is converted into a
#' \code{\link[base]{data.frame}} then passed to
#' \code{\link{as.account.data.frame}}.
#'
#' @param x The \code{\link[base]{matrix}} to convert
#' @param ... Other arguments to pass to \code{\link{as.account}}
#' @return An \code{\link{account}}-class object
#' @seealso \code{\link{as.account.data.frame}}, \code{\link{as.account}},
#'          \code{\link{account}}
#' @export
#' @examples
#' dat <- cbind(c("2010-01-01", "2010-01-02", "2010-01-03"),
#'              c("1000", "-100", "-20"),
#'              c("Initial", "Withdrawal", "Check"))
#' as.account(dat)
as.account.matrix <- function(x, ...) {
  as.account(as.data.frame(x, stringsAsFactors = FALSE), ...)
}
```

`read_csv_account.R`

File containing the function `read_csv_account()`.

```
################################################################################
# read_csv_account.R
# Curtis Miller
# 2020-01-06
################################################################################
# Defining function for reading account data from a CSV file.
################################################################################

################################################################################
# FUNCTIONS
################################################################################

#' Read Account Information from a CSV File
#'
#' Read data to form an \code{\link{account}}-class object from a CSV file.
#'
#' @param file The connection from which data is to be read
#' @param title A character string titling the account
#' @param owner A character string representing the owner of the account
#' @param datecol An identifier for the column of \code{x} representing
#'                transaction dates
#' @param amountcol An identifier for the column of \code{x} representing
#'                  transaction amounts
#' @param memocol An identifier of the column of \code{x} representing
#'                transaction memos
#' @param ... Further arguments to pass to \code{\link[base]{read.csv}}
#' @return An \code{\link{account}}-class object
#' @seealso \code{\link{as.account.data.frame}}, \code{\link{account}}
#' @export
#' @examples
#' input_con <- textConnection(
#' "date, amount, memo
#' 2010-01-01,1000,Initial
#' 2010-01-02,-20,Withdrawal
#' 2010-01-03,-300,Withdrawal"
#' )
#' read_csv_account(input_con)
read_csv_account <- function(file, title = "Account", owner = "noone",
                             datecol = 1, amountcol = 2, memocol = 3, ...) {
  dat <- read.csv(file = file, ...)
  as.account(dat, title = title, owner = owner, datecol = datecol,
             amountcol = amountcol, memocol = memocol)
}
```

`Data.R`

File containing only documentation for the overall package **account** and the data set `accdata`.

```
################################################################################
# Data.R
# Curtis Miller
# 2020-01-06
################################################################################
# Documentation for data sets and the package.
################################################################################

################################################################################
# PACKAGE
################################################################################

#' account: A package for modeling financial accounts
#'
#' The \strong{account} package provides classes and methods for modeling and
#' working with financial accounts, such as a banking account. The core of the
#' package is the associated functions for \code{\link{account}}-class objects,
#' with other functions providing convient summary and plotting features.
#'
#' @docType package
#' @name account-package
NULL

################################################################################
# DATA
################################################################################

#' Example \code{account}-class Object
#'
#' This data set is an \code{\link{account}}-class object for practicing working
#' with these objects. It contains fictitious transactions for a fictitious
#' account.
#'
#' @format An \code{\link{account}}-class object with title "Personal Checking
#'         Account" and owner named Joe Diamond. The account has 35
#'         transactions, starting with an initial balance of 0.55 on 1925-10-08
#'         and ending with a balance of -106.61 on 1925-12-08.
"accdata"
```

After placing these files in `R/` (or modifying when appropriate), rebuild the package. Every major function should now be documented, and `example()` should work as well.

Testing Using testthat

Testing is the process of ensuring that software works as intended. When developing packages, authors should write tests to ensure that important functions work correctly. The package building software can run checks that will run author-written tests, and throw an error if the tests fail. *This is a good thing!* We don't want to distribute code that works incorrectly, and not every bug will automatically reveal itself in an error. Furthermore, extensive testing of even internal functions (not public to the user) can reveal early where problems in code emerged. Writing code without writing high-quality tests is like climbing a cliff without wearing a harness; a small mistake can become a disaster. In fact, programmers may write the tests *before* they write the code, treating the tests as a specification of the software's intended behavior.

We can write automatic tests, and test code lives in the `tests/` subdirectory. Once, R programmers would just put `.R` scripts directly in `tests/`, but there is a package, **testthat**, that helps make writing informative tests easier. (In fact R developers often use it just for assumption checking in general code.) Make sure that you have installed **testthat**. To start using **testthat** for test writing, try loading the **devtools** package and running the command `use_testthat()`. This command will do the following:

1. Create a `tests/testthat/` subdirectory;
2. Add **testthat** to the `Suggests:` field in `DESCRIPTION`; and
3. Create a file `tests/testthat.R` that will run the tests written in `tests/testthat/`.

Do this now in RStudio. These changes will be made, and the file `tests/testthat.R` looks like the following:

```
library(testthat)
library(account)

test_check("account")
```

All test scripts in `tests/testthat/` should be `.R` scripts whose name begins with `test-`. You may assume that any test in the script will load the **testthat** library; calling `library(testthat)` is not necessary (but not an error either). You must have the first command be `context("a description")`; otherwise there' no restrictions, but you will be filling the file with calls to functions from **testthat**. You should also load your package using `library()` in the script; **testthat** won't do this automatically.

When writing a test, we call the function `test_that(s, e)`. `s` is a string describing the tests being conducted in the code block/expression `e`. We often see something like:

```r
test_that("code works right", {
  expectations
})
```

For `expectations` we have a series of calls to `expect` functions. There is a function called `expect()` that can be used to build custom expectations, but most of the time you will be using the following functions:

- `expect_equal(s, e)` to check that `s` is equal to `e` up to some numerical tolerance; for example, `expect_equal(1 + 1, 2)`.
- `expect_identical(s, e)` is like `expect_equal()` except it expects *identical* results. If you're working with numbers, unless you care a great deal about numerical precision, you should use `expect_equal()`.
- `expect_error(s)` checks that `s` throws an error, like `expect_error(1 + "a")`. You can also match for specific error messages, such as `expect_error(1 + "a", "non-numeric argument")`.
- `expect_warning(s)` and `expect_message(s)` is essentially the same as `expect_error()`, but meant for warnings and messages respectively.
- `expect_is(s, e)` checks that `s` inherits from `e`, like `expect_is(1 + 1, "numeric")` or `expect_is(lm(Sepal.Length ~ Species, data = iris), "lm")`.
- `expect_match(s, e)` is for character strings and checks that `s` matches the string `e`, where `e` is a character string interpreted as a regular expression. An example is `expect_match("Hello", "e")`; since `"e"` is in `"Hello"`, we have a match.
- `expect_output(s, e)` works like `expect_match()` except it's capturing the output of the expression `s` and sees if it matches `e`. An example is `expect_output(cat("Hello"), "e")`.
- `expect_silent(s)` simply tests that the expression `s` does not produce any output, including errors, warnings, or messages.

With these `expect` functions, if the expectation is satisfied, nothing happens. If it's not satisfied, an error is produced. When all goes well, the tests are silent.

All told, **testthat** has a grouping of tests. Files contain tests, which themselves contain expectations. Good grouping of expectations produces helpful tests.

I personally like to adopt a file structure in `tests/testthat/` resembling the structure of `R/`, with similarly named files. I group tests based on functions, then put expectations in tests that check important behaviors of the function.

Below are the `test-` files we will use for testing **account**. Copy and paste these files into `tests/testthat/`.

`test-transaction.R`

Let's start with writing tests for transactions. I recommend studying this file closely. In fact, try running this file line-by-line interactively so that you can check that each test passes. Inspect the commands tested both in and out of the expectation. Maybe change a line to a mismatched expecation to see what would happen when an expectation is not met and a test fails.

```r
################################################################################
# test-transaction.R
################################################################################
# Curtis Miller
# 2020-01-09
################################################################################
# Tests for transaction-class objects and their methods
################################################################################

context("transaction objects")

################################################################################
# SETUP
################################################################################

library(account)
trt <- transaction("2010-01-01", 1000, "Initial")
trs <- transaction("2010-01-01", 1000, "Hello")
act1 <- as.account(trs)
act2 <- as.account(trt, title = "My Account", owner = "John Doe")

################################################################################
# CONSTRUCTOR
################################################################################

test_that("transaction constructor works properly", {
  expect_is(trt, "transaction")
  with(trt, {
    expect_is(date, "Date")
    expect_identical(date, as.Date("2010-01-01"))
    expect_equal(amount, 1000)
    expect_match(memo, "Initial")
  })
})

################################################################################
# HELPERS
################################################################################

test_that("is.transaction works properly", {
  expect_true(is.transaction(trt))
  expect_false(is.transaction(data.frame(x = 1:10)))
})

################################################################################
# METHODS
################################################################################

test_that("transaction-class objects print correctly", {
  # The string produces regex string
  expect_output(print(trt),
                paste0(strrep(" ", 4), "2010-01-01:", strrep(" ", 3),
                       "\\+1000\\.00 \\(Initial\\)"))
  expect_output(print(trt, space = 4),
                paste0(strrep(" ", 4), "2010-01-01:", strrep(" ", 1),
                       "\\+1000\\.00 \\(Initial\\)"))
})

test_that("transaction-class objects are correctly turned into accounts", {
  expect_is(act1, "account")
  expect_match(account_title(act1), "Transaction")
  expect_match(account_owner(act1), "noone")
  expect_is(act2, "account")
  expect_match(account_title(act2), "My Account")
  expect_match(account_owner(act2), "John Doe")
  expect_identical(account_dates(act1), as.Date("2010-01-01"))
  expect_identical(account_memos(act1), "Hello")
  expect_identical(account_trans_amounts(act1), 1000)
})
```

`test-transactionHelpers.R`

Next we will write tests explicitly for the helper functions of `transaction`-class objects.

```
################################################################################
# test-transactionHelpers.R
################################################################################
```

```r
# Curtis Miller
# 2020-01-10
################################################################################
# Tests for transaction-class object helper functions
################################################################################

context("transaction-class object helper functions")

################################################################################
# SETUP
################################################################################

library(account)
trt <- transaction("2010-01-01", 1000, "Hello")

################################################################################
# FIELD ACCESS AND MODIFICATION
################################################################################

trt_temp <- trt
test_that("transaction_date() both gets and sets dates", {
  expect_is(transaction_date(trt_temp), "Date")
  expect_identical(transaction_date(trt_temp), as.Date("2010-01-01"))
  expect_identical(transaction_date(trt_temp), trt_temp$date)
  transaction_date(trt_temp) <- "2020-01-01"
  expect_is(transaction_date(trt_temp), "Date")
  expect_identical(transaction_date(trt_temp), as.Date("2020-01-01"))
})

trt_temp <- trt
test_that("amount() both gets and sets amounts", {
  expect_equal(amount(trt_temp), 1000)
  expect_identical(amount(trt_temp), trt_temp$amount)
  amount(trt_temp) <- -20
  expect_equal(amount(trt_temp), -20)
})

trt_temp <- trt
test_that("memo() both gets and sets memos", {
  expect_match(memo(trt_temp), "Hello")
  expect_identical(memo(trt_temp), trt_temp$memo)
  memo(trt_temp) <- "world"
  expect_match(memo(trt_temp), "world")
})
```

`test-account.R`

This file contains basic tests for creating `account`-class objects as well as associated methods. Unfortunately there are no tests for `plot.account()`, since the **testthat** framework is not well equipped for testing plots.

```
################################################################################
# test-account.R
################################################################################
# Curtis Miller
# 2020-01-10
################################################################################
# Tests for account objects and their methods
################################################################################

context("account objects")

################################################################################
# SETUP
################################################################################

library(account)
act1 <- account("2010-01-01", "John Doe")
act2 <- account(start = "2010-01-02", owner = "Jane Doe", init = 100,
                title = "Jane's Account")
act3 <- act2
act3$transactions[[2]] <- transaction("2010-01-05", -20, "")
act3$transactions[[3]] <- transaction("2010-01-03", -30, "")
act4 <- act3
act4$transactions[[4]] <- transaction("2010-01-01", -10, "Error")
act5 <- act4
act5$transactions[[5]] <- transaction("2010-01-01", 0, "Initial")

################################################################################
# CONSTRUCTOR
################################################################################

test_that("account objects are constructed properly", {
  expect_is(act1, "account")
  with(act1, {
    expect_match(title, "Account")
    expect_match(owner, "John Doe")
    expect_is(transactions, "list")
    expect_equal(length(transactions), 1)
    expect_is(transactions[[1]], "transaction")
    with(transactions[[1]], {
      expect_equal(date, as.Date("2010-01-01"))
      expect_equal(amount, 0)
      expect_match(memo, "Initial")
    })
  })
  expect_is(act2, "account")
  with(act2, {
    expect_match(title, "Jane's Account")
    expect_match(owner, "Jane Doe")
    expect_equal(length(transactions), 1)
    expect_is(transactions[[1]], "transaction")
    with(transactions[[1]], {
      expect_equal(amount, 100)
    })
  })
})

################################################################################
# HELPERS
################################################################################

test_that("is.account() works properly", {
  expect_true(is.account(act1))
  expect_false(is.account(data.frame(x = 1:10)))
})

################################################################################
# METHODS
################################################################################

test_that("account objects are sorted correctly", {
  expect_identical(account_dates(sort(act4)),
                   as.Date(c("2010-01-02", "2010-01-01", "2010-01-03",
                             "2010-01-05")))
  expect_identical(account_dates(sort(act4, decreasing = TRUE)),
                   as.Date(c("2010-01-05", "2010-01-03", "2010-01-01",
                             "2010-01-02")))
  expect_identical(account_dates(sort(act5)),
                   as.Date(c("2010-01-01", "2010-01-01", "2010-01-02",
                             "2010-01-03", "2010-01-05")))
  expect_match(account_memos(sort(act5))[[1]], "Initial")
})

test_that("account objects print correctly", {
  # Removing the following since it's hard to copy/paste including whitespace
#   expect_output(print(act3),
#     # To get the following string, I already knew that print() worked as it
#     # should, so I used dput(capture_output(print(act3))) then replaced \n with
#     # new lines and finally escaped +, (, and ) with \\ (which translates to a
#     # single \) since these character have special regular expression meanings
# "Title: Jane's Account
# Owner: Jane Doe
# Transactions:
# ---------------------------------------------------
#     2010-01-02:   \\+100.00 \\(Initial\\)
#     2010-01-05:   -20.00  ")
#     2010-01-03:   -30.00  ")
#   expect_output(print(act3, presort = TRUE),
# "Title: Jane's Account
# Owner: Jane Doe
# Transactions:
# ---------------------------------------------------
#     2010-01-02:   \\+100.00 \\(Initial\\)
#     2010-01-03:   -30.00
#     2010-01-05:   -20.00  ")
})

test_that("Addition of account objects works", {
  act_temp <- act1 + act2
  expect_is(act_temp, "account")
  expect_match(account_title(act_temp), "Account")
  expect_match(account_owner(act_temp), "John Doe")
  expect_identical(account_dates(act_temp),
                   as.Date(c("2010-01-01", "2010-01-02")))
  expect_equal(account_trans_amounts(act_temp), c(0, 100))
})
```

`test-accountHelpers.R`

Next we will write tests explicitly for the helper functions of `account`-class objects. Notice that when testing the private functions they need to be called using `:::`.

```
################################################################################
# test-accountHelpers.R
################################################################################
# Curtis Miller
# 2020-01-12
################################################################################
# Tests for account-class object helper functions
################################################################################
```

```
context("account-class object helper functions")

################################################################################
# SETUP
################################################################################

library(account)
act1 <- account("2010-01-01", "John Doe")
act2 <- account(start = "2010-01-02", owner = "Jane Doe", init = 100,
                title = "Jane's Account")
act3 <- act2
act3$transactions[[2]] <- transaction("2010-01-05", -20, "")
act3$transactions[[3]] <- transaction("2010-01-03", -30, "")
act4 <- act3
act4$transactions[[4]] <- transaction("2010-01-01", -10, "Error")
act5 <- act4
act5$transactions[[5]] <- transaction("2010-01-01", 0, "Initial")

################################################################################
# FIELD ACCESS AND MODIFICATION
################################################################################

act_temp <- act2
test_that("account_title() both gets and sets account titles", {
  expect_match(account_title(act_temp), "Jane's Account")
  expect_identical(account_title(act_temp), act_temp$title)
  account_title(act_temp) <- "Temp"
  expect_match(account_title(act_temp), "Temp")
  account_title(act_temp) <- 19
  expect_match(account_title(act_temp), "19")
})

act_temp <- act2
test_that("account_owner() both gets and sets account owner", {
  expect_match(account_owner(act_temp), "Jane Doe")
  expect_identical(account_owner(act_temp), act_temp$owner)
  account_owner(act_temp) <- "noone"
  expect_match(account_owner(act_temp), "noone")
  account_owner(act_temp) <- 19
  expect_match(account_owner(act_temp), "19")
})

act_temp <- act2
test_that("account_transaction() both gets and sets account transactions", {
  expect_is(account_transactions(act_temp), "list")
  expect_identical(account_transactions(act_temp),
                   list(transaction("2010-01-02", 100, "Initial")))
  expect_identical(account_transactions(act_temp), act_temp$transactions)
  account_transactions(act_temp) <- list(transaction("2020-01-01", 300,
                                         "Initial"))
  expect_identical(account_transactions(act_temp),
                   list(transaction("2020-01-01", 300, "Initial")))
  account_transactions(act_temp)[[2]] <- transaction("2020-01-02", -10, "")
  expect_identical(account_transactions(act_temp)[[2]],
                   transaction("2020-01-02", -10, ""))
})

################################################################################
# PRIVATE FUNCTIONS
################################################################################

act_temp <- act3
test_that("all_transactions() detects all transactions or some not", {
  expect_true(account:::all_transactions(act_temp))
  account_transactions(act_temp)[[4]] <- "Bad!"
  expect_false(account:::all_transactions(act_temp))
})

act_temp <- act4
test_that("bad_Initial() detects no Initial or misplace Initial; FALSE o.w.", {
  expect_true(account:::bad_Initial(act_temp))
  account_transactions(act_temp)[[4]] <- NULL
  expect_false(account:::bad_Initial(act_temp))
  account_transactions(act_temp)[[4]] <- "Bad!"
  expect_error(account:::bad_Initial(act_temp))
  account_transactions(act_temp)[[4]] <- NULL
  account_transactions(act_temp)[[1]] <- NULL
  expect_error(account:::bad_Initial(act_temp))
})

################################################################################
# COLLECTIVE PROPERTIES AND STATISTICS
################################################################################

test_that("account_dates() gets all dates", {
  expect_identical(account_dates(sort(act3)),
                   as.Date(c("2010-01-02", "2010-01-03", "2010-01-05")))
})

test_that("account_trans_amounts() gets all transaction amounts", {
  expect_equal(account_trans_amounts(sort(act3)),
               c(100, -30, -20))
})

test_that("account_memos() gets all account memos", {
  expect_identical(account_memos(sort(act4)), c("Initial", "Error", "", ""))
})

test_that("transaction_count() correctly counts number of transactions", {
  expect_equal(transaction_count(act3), 3)
  expect_identical(transaction_count(act3), length(account_transactions(act3)))
})

################################################################################
# MODIFICATION TOOLS
################################################################################

test_that("account_new_transaction() produces simple accounts", {
  expect_is(account_new_transaction("2010-01-10", -20, "Test"), "account")
  expect_match(account_memos(
                 account_new_transaction("2010-01-10", -20, "Test")), "Test")
  expect_equal(account_trans_amounts(
                 account_new_transaction("2010-01-10", -20, "Test")), -20)
  expect_identical(account_dates(
                     account_new_transaction("2010-01-10", -20, "Test")),
                   as.Date("2010-01-10"))
  expect_match(account_title(
                 account_new_transaction("2010-01-10", -20, "Test")),
               "Transaction")
  expect_match(account_owner(
                 account_new_transaction("2010-01-10", -20, "Test")),
               "noone")
})

test_that("account_delete_transaction() deletes transactions", {
  expect_error(account_delete_transaction(act4))
  expect_warning(account_delete_transaction(act4, date = "2100-01-01",
                                            memo = "Turkey"))
  expect_warning(account_delete_transaction(act4, date = "2100-01-01"))
  expect_warning(account_delete_transaction(act4, memo = "Turkey"))
  expect_warning(account_delete_transaction(act4, date = "2010-01-02",
                                            memo = "Error"))
  # If warnings are not suppressed, this will be a problem
  expect_identical(suppressWarnings(
                     account_delete_transaction(act4, date = "2010-01-02",
                                                    memo = "Error")), act4)
  act_temp <- act4
  account_transactions(act_temp) <- account_transactions(act_temp)[-4]
  expect_identical(account_delete_transaction(act4, memo = "Error"),
                   act_temp)
  act_temp <- act4
  account_transactions(act_temp) <- account_transactions(act_temp)[-(2:3)]
  expect_identical(account_delete_transaction(act4, memo = ""),
                   act_temp)
  act_temp <- act5
  account_transactions(act_temp) <- account_transactions(act_temp)[1:3]
  expect_identical(account_delete_transaction(act5, date = "2010-01-01"),
                   act_temp)
  act_temp <- act5
  account_transactions(act_temp) <- account_transactions(act_temp)[-5]
  expect_identical(account_delete_transaction(act5, date = "2010-01-01",
                                              memo = "Initial"),
                   act_temp)
})
```

`test-summary.account.R`

This file contains basic tests for creating `summary.account`-class objects as well as associated methods. `summary.account`-class objects are produced by the generic function `summary()`, so the function is both a method and an object constructor.

```
################################################################################
# test-summary.account.R
################################################################################
# Curtis Miller
# 2020-01-12
################################################################################
# Tests for summary.account-class objects
################################################################################

context("summary.account objects")

################################################################################
# SETUP
################################################################################

library(account)

act <- account(start = "2010-01-02", owner = "Jane Doe", init = 100,
               title = "Jane's Account")
```

```r
act2 <- act
act2$transactions[[2]] <- transaction("2010-01-05", -20, "")
act2$transactions[[3]] <- transaction("2010-01-03", -30, "")
act3 <- act2
act3$transactions[[4]] <- transaction("2010-01-01", -10, "Error")
act4 <- act3
act4$transactions[[5]] <- transaction("2010-01-01", 0, "Initial")
act4$transactions[[6]] <- transaction("2010-01-10", 21, "Deposit")

################################################################################
# CONSTRUCTOR
################################################################################

test_that("summary.account() works properly", {
  expect_warning(summary(act3))
  expect_error(summary(account_new_transaction("2010-01-01", 0)))
  expect_equal(length(summary(act4)$rtrans), 5L)
  act_sum <- summary(act4, recent = 2)
  expect_is(act_sum, "summary.account")
  expect_match(act_sum$title, "Jane's Account")
  expect_match(act_sum$owner, "Jane Doe")
  expect_equal(act_sum$balance, 61)
  expect_equal(act_sum$tcount, 6L)
  expect_equal(length(act_sum$rtrans), 2)
  expect_true(all(sapply(act_sum$rtrans, class) == "transaction"))
  expect_identical(act_sum$rtrans, account_transactions(act4)[c(6, 2)])
})

################################################################################
# METHODS
################################################################################

test_that("account-class object summaries print correctly", {
  act_sum <- summary(act4, recent = 2)
  # Removing the following since it's hard to copy/paste including whitespace
#   expect_output(print(act_sum),
# "
# \\tJane's Account
#
# Owner:              Jane Doe
# Transactions:              6
# Balance:               61.00
#
# Recent Transactions:
# ----------------------------------------------------
#     2010-01-10:     \\+21.00 \\(Deposit\\)
#     2010-01-05:     -20.00  ")
})
```

test-Generics.R

Here we write tests for our generic functions. There is only one generic function, `as.account()`, and generic functions usually don't do much; they only call `UseMethod()`. Thus we're not testing the generic function so much as we're testing the methods associated with the generic function.

```
################################################################################
# test-Generics.R
################################################################################
# Curtis Miller
# 2020-01-13
################################################################################
# Tests for generic functions
################################################################################

context("Generic functions")

################################################################################
# SETUP
################################################################################

library(account)

mat1 <- cbind(date = c("2010-01-01", "2010-01-02"),
              amount = c("100", "-20"),
              memo = c("Initial", ""))
mat2 <- mat1[, c(2, 1, 3)]
df1 <- as.data.frame(mat1, stringsAsFactors = FALSE)
df2 <- as.data.frame(mat2, stringsAsFactors = FALSE)
t_list <- list(transaction("2010-01-01", 100, "Initial"),
               transaction("2010-01-02", -20))
acc <- account("2010-01-01", owner = "noone", title = "Account")
account_transactions(acc) <- t_list

################################################################################
# DATA FRAME TO ACCOUNT
################################################################################

acc_temp <- acc
test_that("as.account() converts data frames to account-class objects", {
  expect_identical(as.account(df1), acc_temp)
  expect_identical(as.account(df2, datecol = 2, amountcol = 1, memocol = 3),
                   acc_temp)
  expect_identical(as.account(df2, datecol = "date", amountcol = "amount",
                              memocol = "memo"), acc_temp)
  account_title(acc_temp) <- "TA"
  account_owner(acc_temp) <- "Jack"
  expect_identical(as.account(df1, title = "TA", owner = "Jack"), acc_temp)
})

acc_temp <- acc
test_that("as.account() converts matrix to account-class objects", {
  expect_identical(as.account(mat1), acc_temp)
  expect_identical(as.account(mat2, datecol = 2, amountcol = 1, memocol = 3),
                   acc_temp)
  expect_identical(as.account(mat2, datecol = "date", amountcol = "amount",
                              memocol = "memo"), acc_temp)
  account_title(acc_temp) <- "TA"
  account_owner(acc_temp) <- "Jack"
  expect_identical(as.account(mat1, title = "TA", owner = "Jack"), acc_temp)
})
```

test-read_csv_account.R

Finally we test `read_csv_account()`. This should be able to read a file from disk and convert it to an `account`-class object. While it should be possible to provide a file for testing, we will content ourselves with working with a string connection.

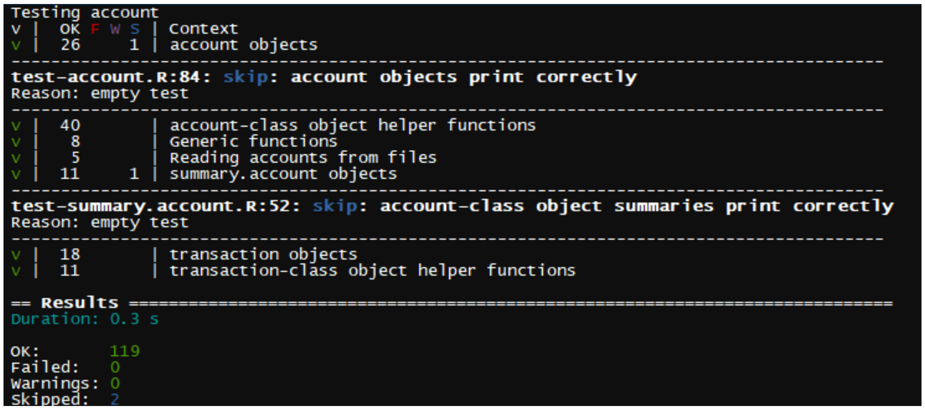
```
################################################################################
# test-read_csv_account.R
# Curtis Miller
# 2020-01-13
################################################################################
# Testing functions for reading account objects
################################################################################

context("Reading accounts from files")

################################################################################
# SETUP
################################################################################

library(account)

input_str1 <- "date,amount,memo
2010-01-01,1000,Initial
```

```
2010-01-02,-20,Withdrawal
2010-01-03,-300,Withdrawal"

input_str2 <- "memo,date,amount
Initial,2010-01-01,1000
Withdrawal,2010-01-02,-20
Withdrawal,2010-01-03,-300"

acc <- account(title = "Account", owner = "noone", start = "2010-01-01")
account_transactions(acc) <- list(
  transaction("2010-01-01", 1000, "Initial"),
  transaction("2010-01-02", -20, "Withdrawal"),
  transaction("2010-01-03", -300, "Withdrawal")
)

################################################################################
# READING FILES
################################################################################

test_that("read_csv_account() can read CSV files", {
  expect_identical(read_csv_account(textConnection(input_str1)), acc)
  expect_identical(read_csv_account(textConnection(input_str2), datecol = 2,
                                    amountcol = 3, memocol = 1), acc)
  expect_identical(read_csv_account(textConnection(input_str2),
                                    datecol = "date", amountcol = "amount",
                                    memocol = "memo"), acc)
  expect_match(account_title(read_csv_account(textConnection(input_str1),
                             title = "test")), "test")
  expect_match(account_owner(read_csv_account(textConnection(input_str1),
                             owner = "Joe")), "Joe")
})
```

Once these files are in the right subdirectory, you can run the tests using `devtools::test()` or the shortcut keys `Ctrl+Shift+T`. These will cause the tests to execute, and in one of the RStudio panes you'd see the following output:

These are the results of the tests. The output says how many checks passed, how many didn't pass, and how many were skipped. If a test was passed, there will be output explaining why it didn't pass and, if possible, how far from expectations the resulting output was. In this case, some tests were skipped since their code was commented out (they were tests for printing), and the output reports which were skipped.

Writing tests and developing test cases is an art of its own. Just because your tests pass does not mean there are not errors in the code; it just means your tests can't detect them. Good testing, though, can save you many hours of



```
Testing account
v |  OK F W S | Context
v |  26     1 | account objects
--------------------------------------------------------------------------------------
test-account.R:84: skip: account objects print correctly
Reason: empty test
--------------------------------------------------------------------------------------
v |  40       | account-class object helper functions
v |   8       | Generic functions
v |   5       | Reading accounts from files
v |  11     1 | summary.account objects
--------------------------------------------------------------------------------------
test-summary.account.R:52: skip: account-class object summaries print correctly
Reason: empty test
--------------------------------------------------------------------------------------
v |  18       | transaction objects
v |  11       | transaction-class object helper functions

== Results ============================================================================
Duration: 0.3 s

OK:        119
Failed:    0
Warnings:  0
Skipped:   2
```

Figure 17: Testing

headache figuring out when and why your code went wrong. You don't need test code until your code breaks without you knowing it, so I recommend taking testing seriously. I believe non-professional programmers, in particular, are too lax when it comes to testing and assumption checking, and the result is hours of wasted time tracking down bugs. In any sufficiently complex project (and it doesn't take much for a project to become "sufficiently" complicated) testing is crucial.

Lecture 7

Statistical Intervals

A **statistical interval** is an interval computed using a sample intended to capture some aspect of the population with some user-specified probability. In this lecture I will discuss some statistical intervals and demonstrate how to compute these intervals in R. When discussing intervals, remember that these include *bounds*, which is viewed as an interval with one end point being infinite.

Every statistics student encounters confidence intervals and statistics instructors drill students on what the proper interpretation of a confidence interval is. Statisticians care so much about the interpretation of the confidence interval because not only is the wrong interpretation simply wrong, there probably is another interval with the mistaken interpretation that is computed differently from the confidence interval. Here we see some of those other intervals.

Confidence Intervals

Every statistics student should learn how to compute the most common statistical interval: the confidence interval. A $100C\%$ **confidence interval (CI)** for some parameter θ is an interval such that the probability the interval captures the unknown parameter θ when computed from a random sample is C. You should already know how to compute confidence intervals for many contexts, and I will not repeat the discussion here.

Credible Intervals

The Bayesian $100C\%$ **credible interval** is an interval such that the probability a population parameter θ (here viewed as a random variable of its own) is a specified C. Note that this is *not* the same interpretation as the confidence intervals, with the subtle difference being that for credible intervals θ is viewed as random while for confidence intervals θ is fixed and non-raondom, so talking about the "probability" the parameter is in a fixed interval doesn't make sense (it's either 1 or 0 depending on whether the parameter is or is not in the interval, though the researcher doesn't know which is the case).

Computing Bayesian credible intervals requires adopting the Bayesian prior/posterior mindset that goes beyond the scope of this course, so I will not discuss these intervals. Furthermore, Bayesians will compute all of the following intervals differently than the frequentists, so we will ignore Bayesianism from now on.

Prediction Intervals

A $100C\%$ **prediction interval (PI)** is an interval such that, if the interval is computed from a random sample $X_1, \dots, X_n$, the probability the interval includes the observation X_{n+1} is C. In short, it's an interval that predicts where a future observation will appear based on existing observations. Contrast this with CIs which only try to describe the uncertainty associated with the value of a parameter θ; CIs predict *nothing*.

Two comments on PIs: first, distributional assumptions matter more to PIs than to CIs. Asymptotic theory allows us to approximate distributions of statistics with some limiting distribution (for example, approximate the distribution of the sample mean with the Normal distribution), but since we're concerned with the behavior of a single observation, we can't neglect that observation's distribution and substitute a limiting distribution like we could with a statistic's distribution. Second, while prediction intervals can narrow as we collect more data, these intervals don't "go to zero," unlike confidence intervals, since an individual observation always has some natural variation not going to zero that must be accounted for.

Let's start, for instance, with constructing a prediction interval for an observation from a Normal distribution with known standard deviation σ. Throughout this lecture assume data is *i.i.d.*. Then $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i \sim N\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$. Also, $X_{n+1} \sim N(0, \sigma)$, and since $\bar{X}$ and X_{n+1} are independent of each other, $X_{n+1} - \bar{X} \sim N\left(0, \sigma\sqrt{1 + \frac{1}{n}}\right)$. Thus we can say

$$\frac{X_{n+1} - \bar{X}}{\sigma\sqrt{1 + \frac{1}{n}}} \sim N(0,1).$$

We can then form an equal-tail interval by noticing:

$$P\left(-z^* \leq \frac{X_{n+1} - \bar{X}}{\sigma\sqrt{1 + \frac{1}{n}}} \leq z^*\right) = P\left(\bar{X} - z^*\sigma\sqrt{1 + \frac{1}{n}} \leq X_{n+1} \leq \bar{X} + z^*\sigma\sqrt{1 + \frac{1}{n}}\right) = C$$

where $z^* = z_{1-C/2}$. In short, the equal-tailed PI is $\bar{x} \pm z^*\sigma\sqrt{1 + \frac{1}{n}}$.

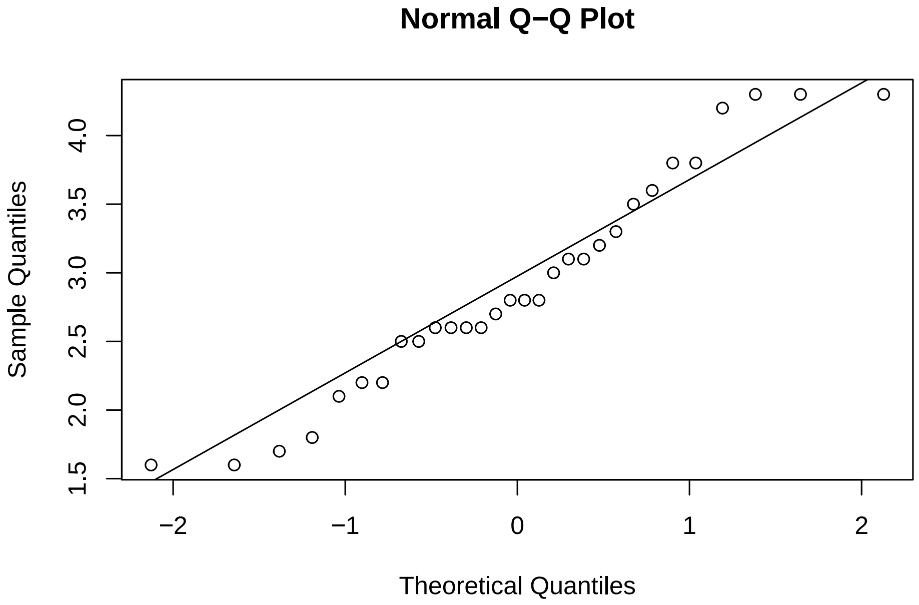
Suppose that σ is not known. The resulting calculation is similar to that listed above, and the resulting interval is $\bar{x} \pm t^* s\sqrt{1 + \frac{1}{n}}$, where s is the sample standard deviation and $t^* = t_{n-1,1-C/2}$ is the corresponding critical value of the t-distribution with $n - 1$ degrees of freedom.

Below is a function for computing the prediction interval for Normally distributed data. It also allows for one-sided intervals, and uses an interface resembling t-test.

```r
pi_norm <- function(x, conf.level = 0.95, alternative = "two.sided") {
  alpha <- 1 - conf.level
  n <- length(x)
  xbar <- mean(x)
  err <- sd(x) * sqrt(1 + 1 / n)
  crit <- switch(alternative,
                 "two.sided" = qt(alpha / 2, df = n - 1, lower.tail = FALSE),
                 "less" = -qt(alpha, df = n - 1, lower.tail = FALSE),
                 "greater" = qt(alpha, df = n - 1, lower.tail = FALSE),
                 # Below is the "default" switch, triggered if none of the above
                 stop("alternative must be one of two.sided, less, greater"))
  interval <- switch(alternative,
                     "two.sided" = c(xbar - crit * err, xbar + crit * err),
                     "less" = c(xbar + crit * err, Inf),
                     "greater" = c(-Inf, xbar + crit * err),
                     stop("How did I get here?"))
  attr(interval, "conf.level") <- conf.level
  interval
}
```

Let's demonstrate by predicting the weight of a cabbage. The data set `cabbages` (**MASS**) contains data from a cabbage field trial (in kilograms). Here I collect the weights of cabbages from one cultivator.

```r
library(MASS)
cabbage_weight <- subset(cabbages, subset = Cult == "c39")$HeadWt
qqnorm(cabbage_weight)
qqline(cabbage_weight)
```



```
pi_norm(cabbage_weight, conf.level = 0.9)
```

```
## [1] 1.516780 4.296554
## attr(,"conf.level")
## [1] 0.9
```

The procedure above, though, is for Normal data. *It would never be appropriate to use it for non-Normal data, not even for large sample sizes.* So what should we do when our data is not Normally distributed? We have two options:

1. If we know the distribution the data came from, we should use an interval designed for that distribution. This is the parameteric approach.
2. Use a procedure that assumes very little about the distribution. This is the non-parametric approach.

Computing nonparametric intervals can be done if one assumes the data was drawn from a continuous distribution (it can be done if this not true too but the math is much harder due to the need to account for ties) using the procedure listed here. These intervals are not exact prediction intervals since you may not be able to get an interval for the exact confidence level specified, but they involve few assumptions and we should be able to get close to our desired confidence level. These intervals use the order statistics of the data for their bounds.

The function below can conservatively estimate a prediction interval using a suggested confidence rate. (Only two-sided intervals are allowed by this code.)

```
no_param_pi <- function(x, conf.level = 0.95) {
  n <- length(x)
  x <- sort(x)
  j <- max(floor((n + 1) * (1 - conf.level) / 2), 1)
  conf.level <- (n + 1 - 2 * j)/(n + 1)
  interval <- c(x[j], x[n + 1 - j])
  attr(interval, "conf.level") <- conf.level
  interval
}
```

Notice the result of the following:

```
no_param_pi(rnorm(1000))
```

```
## [1] -1.983822  1.939687
## attr(,"conf.level")
## [1] 0.95005
```

The confidence level is slightly higher than the specified 95% but close enough; being lower can happen if there's not enough data. Furthermore, the resulting prediction interval is close to the interval $(-2, 2)$; recall that the probability that a Normal random variable is within two standard deviations of its mean is roughly 95%, so this interval appears to function as it should.

Here we use it on the cabbage data:

```
no_param_pi(cabbage_weight)
```

```
## [1] 1.6 4.3
## attr(,"conf.level")
## [1] 0.9354839
```

Due to the limitations of the data set, we don't get a 95% confidence interval; the procedure was forced to just use the maximum and the minimum of the data set as the prediction interval. Nevertheless, it's close to a 95% interval.

Replication Intervals

Prediction intervals need not be just for a single observation; they could be for any quantity we estimate from future data. For example, we could compute a prediction interval for a future sample mean, of any sample size. When we set the sample size of the future sample mean equal to the sample size of the computed sample mean, I like to call the resulting prediction interval a $100C\%$ **replication interval (RI)** since this interval can be interpreted as an interval attempting to capture the value of sample means in replication studies.

Let's revisit again the Normal case. Let $\bar{X}_{1,n} = \frac{1}{n}\sum_{i=1}^{n} X_i$ represent the observed sample mean and $\bar{X}_{2,m} = \frac{1}{m}\sum_{i=n+1}^{n+m} X_i$ the future sample mean for a sample of size m; both of these are treated as random variables, but to construct the interval we will be computing $\bar{X}_{1,n}$. Since $\bar{X}_{1,n} \sim N\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$ and $\bar{X}_{2,m} \sim N\left(\mu, \frac{\sigma}{\sqrt{m}}\right)$ and the two random variables are independent, then

$$\frac{\bar{X}_{1,n} - \bar{X}_{2,m}}{\sigma\sqrt{\frac{1}{n} + \frac{1}{m}}} \sim N(0,1).$$

And in fact the distributional logic developed above applies to this random variable, so we can get a confidence interval even when σ is not known, and obtain the interval $\bar{x} \pm t^* s\sqrt{\frac{1}{n} + \frac{1}{m}}$ for predicting the value of $\bar{X}_{2,m}$, with t^* as before. In fact, if we say $m = n$, then our interval will be $\bar{x} \pm \sqrt{2}t^* \frac{s}{\sqrt{n}}$.

Since these intervals are based on the behavior of averages, we can say not only that these intervals will go to zero as we increase our sample sizes m and n, but also that the intervals should work well regardless of the underlying data distribution if both m and n are sufficiently large. These are unlike the earlier prediction intervals, but as with confidence intervals, this is because the objects under study are averages.

Below is a function for computing these replication intervals.

```r
repint <- function(x, m = length(x), conf.level = 0.95,
                   alternative = "two.sided") {
  alpha <- 1 - conf.level
  n <- length(x)
  xbar <- mean(x)
  err <- sd(x) * sqrt(1 / m + 1 / n)
  crit <- switch(alternative,
                 "two.sided" = qt(alpha / 2, df = n - 1, lower.tail = FALSE),
                 "less" = -qt(alpha, df = n - 1, lower.tail = FALSE),
                 "greater" = qt(alpha, df = n - 1, lower.tail = FALSE),
                 # Below is the "default" switch, triggered if none of the above
                 stop("alternative must be one of two.sided, less, greater"))
  interval <- switch(alternative,
                     "two.sided" = c(xbar - crit * err, xbar + crit * err),
                     "less" = c(xbar + crit * err, Inf),
                     "greater" = c(-Inf, xbar + crit * err),
                     stop("How did I get here?"))
  attr(interval, "conf.level") <- conf.level
  interval
}
```

Here we apply the interval to the cabbage data:

```r
repint(cabbage_weight)
```

```
## [1] 2.481725 3.331609
## attr(,"conf.level")
## [1] 0.95
```

Translated into English, we would say that, with 95% confidence, a future mean in a replication study (using the same sample size) should lie between `r round(repint(cabbage_weight)[[1]], digits = 2)` and `r round(repint(cabbage_weight)[[2]], digits = 2)`.

Tolerance Intervals

A **tolerance interval (TI)** is an interval that contains at least $100K\%$ of the population with $100C\%$ confidence. Let's unpack that statement. The objective of the interval is to create an interval that will contain not only a single future observation but $100K\%$ of *all* observations (present and future) for some user-selected $K \in (0, 1)$. We wish to get such an interval with $100C\%$ confidence, with C being the user-specified confidence level that's appeared in the other intervals. This interval, however, will likely be wider than the true population interval that contains $100K\%$ of population values due to measurement error, and as a result we should say that the resulting interval contains *at least* $100K\%$ of the population values with our specified confidence level.

Unlike PIs, TIs are not trying to predict a future observation but estimate an interval. They're interval estimates for intervals. But like PIs, TIs can be highly parametric (with distributional assumptions mattering even for large sample sizes) and should not be expected to go to zero for large sample sizes since the underlying interval they're trying to capture isn't length zero itself.

The R package **tolerance** provides functions for computing both parametric and nonparametric tolerance intervals. For Normal data, we can use the function `normtol.int()`. Let's demonstrate by computing a tolerance interval for cabbage weight, requiring that the interval capture 90% of cabbage weights. We will make this a 99% confidence interval.

```r
library(tolerance)
normtol.int(cabbage_weight, alpha = 1 - 0.99, P = 0.9, side = 2)
```

```
##   alpha   P    x.bar 2-sided.lower 2-sided.upper
## 1  0.01 0.9 2.906667     0.9813193      4.832014
```

(Note the parameter `alpha` and `P`, which correspond to $1-C$ and K in the above presentation, respectively.) Additionally, we can compute nonparametric TIs using the function `nptol.int()`. Like with nonparametric PIs, nonparametric TIs depend heavily on sample quantiles. (This is in fact an important theme in nonparametric statistics in general.)

```r
nptol.int(cabbage_weight, alpha = 1 - 0.99, P = 0.9, side = 2)
```

```
##   alpha   P 2-sided.lower 2-sided.upper
## 1  0.01 0.9           1.6           4.3
```

Lecture 8

Linear Regression Models

A **linear regression model** is a statistical model of the form:

$$y_i = \beta_0 + \beta_1 x_{1i} + \cdots + \beta_k x_{ki} + \epsilon_i.$$

The term y_i is the **response variable** or **dependent variable**, while the variables $x_{1i}, \ldots, x_{ki}$ are called **explanatory variables**, **independent variables**, or **regressors**. The terms $\beta_0, \beta_1, \ldots, \beta_k$ are called the **coefficients** of the model, and ϵ_i is the **error** or **residual** term. The term β_0 is often called the **intercept** of the model since it's the predicted value of y_i when all the explanatory variables are zero.

Simple linear regression refers to a linear regression model with only one explanatory variable:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i.$$

This model corresponds to the one-dimensional lines all math students studied in basic algebra, and it's easily visualized. Additionally, when estimating the model via least-squares estimation, it's easy and intuitive to write the answer. (Actually the least-squares solution for the full model isn't hard to write either after one learns some linear algebra, but matrices may scare novice statistics students.) However, the model is generally too simple and real-world applications generally include more than one regressor.

Generally statisticians treat the term ϵ_i as the only source of randomness in linear regression models; regressors may or may not be random. Thus most assumption concern the behavior of ϵ_i, and include assuming that $E[\epsilon_i] = 0$ and $\text{Var}(\epsilon_i) = \sigma^2$. We may say more about the errors, such as that the errors are *i.i.d.* and maybe Normally distributed.

Estimating Simple Linear Regression Models

The models listed above are population models. Our objective is to estimate the parameters of these models. Let $\hat{\beta}_0$ and $\hat{\beta}_1$ denote estimates of β_0 and β_1, respectively. The **predicted** value of y given an input x is:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x.$$

We call $\hat{\epsilon}_i = y_i - \hat{y}_i$ the **estimated residual** of the model. The question now is how one should estimate the coefficients of the model. The most popular approach is **least-squares minimization**, where $\hat{\beta}_0$ and $\hat{\beta}_1$ are picked such that the sum of square errors:

$$SSE(\hat{\beta}_0, \hat{\beta}_1) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

is minimized. Suppose r is the sample correlation, s_x is the sample standard devitation of the explanatory variable, s_y the sample standard deviation of the response variable, $\bar{x}$ the sample mean of the explanatory variable, and $\bar{y}$ the sample mean of the response variable. Then we have a simple solution for the estimates of the coefficients of the least-squares line:

$$\hat{\beta}_1 = r\frac{s_y}{s_x}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}.$$

Practitioners also often wish to estimate the variance of the residuals, σ^2, and do so using $\hat{\sigma}^2 = \frac{1}{n-2} SSE(\hat{\beta}_1, \hat{\beta}_2)$. The term $n - 2$ is the **degrees of freedom** of the simple linear regression model.

When estimating using least squares, the predicted value $\hat{y}$ can be interpreted as the expected value of the response variable y given the value of the explanatory variable x. In fact, the predicted value of y at the mean of the explanatory variables $\bar{x}$ is $\bar{y}$. Regression models, however, don't have to be estimated using the least-squares principle; for example, we could adopt the least absolute deviation principle, and the interpretation of $\hat{y}$ would change if we did so. Additionally, the coefficients of the model get interpretations; we say that if we increase an input explanatory variable x by one unit, then we expect y to be β_1 units higher. (The intercept term is generally less interesting in regression models, but its interpretation is clear; it's the expected value of y when $x = 0$.) It's because of interpretations like these that we may **rescale** the data, replacing x_i with $\frac{x_i - \bar{x}}{s_x}$ and y_i with $\frac{y_i - \bar{y}}{s_y}$. Then we would interpret β_1 as being how many *standard deviations* y changes by when we change x by one *standard deviation.* This is a more universally interpretable model.

Linear models are in fact quite expressive; many interesting models can be written as linear models. For instance, consider the model:

$$y_i = \gamma_0 \gamma_1^{x_i} \eta_i.$$

In this model, η_i is the multiplicative residual, and is assumed to be non-negative with mean 1. If we take the natural log of both sides of this equation, we obtain the mode:

$$\ln(y_i) = \ln(\gamma_0) + \ln(\gamma_1)x_i + \ln(\eta_i).$$

This is in fact the linear model we've already been considering, but one for $\ln(y_i)$ rather than y_i directly. This is known as a **log transformation**, a useful and popular trick. The interpretations of the coefficients change, though: a unit increase in the exogenous variable causes us to expect a β_1% change in y_i. There's also the model:

$$y_i = \beta_0 + \beta_1 \ln(x_i) + \epsilon_i.$$

In this model a 1% change in the regressor leads to an expected increase of β_1 of the response. There's also the model:

$$\ln(y_i) = \beta_0 + \beta_1 \ln(x_i) + \epsilon_i.$$

Here, a 1% increase in x_i would cause us to expect a β_1% increase in y_i. These interpretations matter, and are a part of model building.

In R, linear models are estimated using the function `lm()`. If `x` is the explanatory variable and `y` the response variable and both are in a data set `d`, we estimate the model using `lm(y ~ x, data = d)`. Generally we save the results of the fit for later use.

Let's estimate a linear regression model for the sepal length of iris flowers using the sepal width. We can do so with the following:

```
(fit <- lm(Sepal.Length ~ Sepal.Width, data = iris))
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Sepal.Width, data = iris)
##
## Coefficients:
## (Intercept)  Sepal.Width
##      6.5262      -0.2234
```

We don't need to explicitly tell R to include an intercept term; `lm()` automatically includes one. In general you should include intercept terms in your regression models even if the theoretical model you're estimating does not have an intercept term, since doing so helps ensure the residuals of the model have mean zero. However, if you really want to exclude the intercept term (and thus be fitting the model $y_i = \beta x_i + \epsilon_i$), you can suppress intercept term estimation like so:

```
lm(Sepal.Length ~ Sepal.Width - 1, data = iris)
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Sepal.Width - 1, data = iris)
##
## Coefficients:
## Sepal.Width
##       1.869
```

R will print out a basic summary of a `lm()` fit when the fit object (an object of class `lm`) is printed. We can see more information using `summary()`.

```
summary(fit)
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Sepal.Width, data = iris)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.5561 -0.6333 -0.1120  0.5579  2.2226
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   6.5262     0.4789   13.63   <2e-16 ***
## Sepal.Width  -0.2234     0.1551   -1.44    0.152
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.8251 on 148 degrees of freedom
## Multiple R-squared:  0.01382,    Adjusted R-squared:  0.007159
## F-statistic: 2.074 on 1 and 148 DF,  p-value: 0.1519
```

We will discuss later the meaning of this information. That said, while this information is nice, it's not… pretty. There is a package called **stargazer**, though, that is meant for making nice presentations of regression results.

```
library(stargazer)
```

```
##
## Please cite as:
```

```
##  Hlavac, Marek (2018). stargazer: Well-Formatted Regression and Summary Statistics Tables.
##  R package version 5.2.2. https://CRAN.R-project.org/package=stargazer
```

```r
stargazer(fit, type = "text")
```

```
## 
## ===============================================
##                         Dependent variable:    
##                     ---------------------------
##                            Sepal.Length        
## -----------------------------------------------
## Sepal.Width                   -0.223           
##                               (0.155)          
##                                                
## Constant                     6.526***          
##                               (0.479)          
##                                                
## -----------------------------------------------
## Observations                    150            
## R2                             0.014           
## Adjusted R2                    0.007           
## Residual Std. Error      0.825 (df = 148)      
## F Statistic            2.074 (df = 1; 148)     
## ===============================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

Actually, `stargazer()` can print out these regression tables in other formats, particularly LaTeX and HTML. This makes inserting these tables into other documents (specifically, papers) easier. (If you want to put your table into a word processor such as Microsoft Word, try HTML format, and save in an HTML document; the text processor may be able to convert the HTML table into something it understands. Or you could just man up and learn LaTeX; it looks better anyway.)

```r
stargazer(fit)  # Default is LaTeX
```

```
## 
## % Table created by stargazer v.5.2.2 by Marek Hlavac, Harvard University. E-mail: hlavac at fa
## % Date and time: Fri, Feb 14, 2020 - 05:42:03 PM
## \begin{table}[!htbp] \centering 
##   \caption{} 
##   \label{} 
## \begin{tabular}{@{\extracolsep{5pt}}lc} 
## \\[-1.8ex]\hline 
## \hline \\[-1.8ex] 
##  & \multicolumn{1}{c}{\textit{Dependent variable:}} \\ 
## \cline{2-2} 
## \\[-1.8ex] & Sepal.Length \\
## \hline \\[-1.8ex]
##  Sepal.Width & $-$0.223 \\
##   & (0.155) \\
##   & \\
##  Constant & 6.526$^{***}$ \\
##   & (0.479) \\
##   & \\
## \hline \\[-1.8ex]
## Observations & 150 \\
## R$^{2}$ & 0.014 \\
## Adjusted R$^{2}$ & 0.007 \\
## Residual Std. Error & 0.825 (df = 148) \\
## F Statistic & 2.074 (df = 1; 148) \\
## \hline
## \hline \\[-1.8ex]
## \textit{Note:}  & \multicolumn{1}{r}{$^{*}$p$<$0.1; $^{**}$p$<$0.05; $^{***}$p$<$0.
## \end{tabular}
## \end{table}
```

```r
stargazer(fit, type = "html")
```

```
##
## <table style="text-align:center"><tr><td colspan="2" style="border-bottom: 1px soli
## <tr><td></td><td colspan="1" style="border-bottom: 1px solid black"></td></tr>
## <tr><td style="text-align:left"></td><td>Sepal.Length</td></tr>
## <tr><td colspan="2" style="border-bottom: 1px solid black"></td></tr><tr><td style=
## <tr><td style="text-align:left"></td><td>(0.155)</td></tr>
## <tr><td style="text-align:left"></td><td></td></tr>
## <tr><td style="text-align:left">Constant</td><td>6.526<sup>***</sup></td></tr>
## <tr><td style="text-align:left"></td><td>(0.479)</td></tr>
## <tr><td style="text-align:left"></td><td></td></tr>
## <tr><td colspan="2" style="border-bottom: 1px solid black"></td></tr><tr><td style=
## <tr><td style="text-align:left">R<sup>2</sup></td><td>0.014</td></tr>
## <tr><td style="text-align:left">Adjusted R<sup>2</sup></td><td>0.007</td></tr>
## <tr><td style="text-align:left">Residual Std. Error</td><td>0.825 (df = 148)</td></
## <tr><td style="text-align:left">F Statistic</td><td>2.074 (df = 1; 148)</td></tr>
## <tr><td colspan="2" style="border-bottom: 1px solid black"></td></tr><tr><td style=
## </table>
```

There's a lot of `stargazer()` parameters you can modify to get just the right table you want, and I invite you to read the function's documentation for yourself if you're planning on publishing your regression models; for now it makes no sense to discuss this when we're not even sure what we're modifying!

That said, we may want specific information from our model. `lm()` returns a list with the class `lm`, and we can examine this list to see what information is already available to us:

```r
str(fit)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 6.526 -0.223
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "Sepal.Width"
##  $ residuals    : Named num [1:150] -0.644 -0.956 -1.111 -1.234 -0.722 ...
##   ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:150] -71.566 -1.188 -1.081 -1.187 -0.759 ...
##   ..- attr(*, "names")= chr [1:150] "(Intercept)" "Sepal.Width" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:150] 5.74 5.86 5.81 5.83 5.72 ...
##   ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:150, 1:2] -12.2474 0.0816 0.0816 0.0816 0.0816 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:150] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "Sepal.Width"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.08 1.02
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 148
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = Sepal.Length ~ Sepal.Width, data = iris)
##  $ terms        :Classes 'terms', 'formula'  language Sepal.Length ~ Sepal.Width
##   .. ..- attr(*, "variables")= language list(Sepal.Length, Sepal.Width)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "Sepal.Length" "Sepal.Width"
##   .. .. .. ..$ : chr "Sepal.Width"
##   .. ..- attr(*, "term.labels")= chr "Sepal.Width"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(Sepal.Length, Sepal.Width)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "Sepal.Length" "Sepal.Width"
##  $ model        :'data.frame':   150 obs. of  2 variables:
##   ..$ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##   ..$ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language Sepal.Length ~ Sepal.Wi
##   .. .. ..- attr(*, "variables")= language list(Sepal.Length, Sepal.Width)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "Sepal.Length" "Sepal.Width"
##   .. .. .. .. ..$ : chr "Sepal.Width"
##   .. .. ..- attr(*, "term.labels")= chr "Sepal.Width"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(Sepal.Length, Sepal.Width)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "Sepal.Length" "Sepal.Width"
##  - attr(*, "class")= chr "lm"
```

One thing we may want is the model's coefficients: we can obtain them using the `coef()` function (or use, say, `fit$coefficients`).

```r
coef(fit)
```

```
## (Intercept) Sepal.Width
##   6.5262226  -0.2233611
```

We may also want the residuals of the model; we can get those using `resid()` or `fit$residuals`.

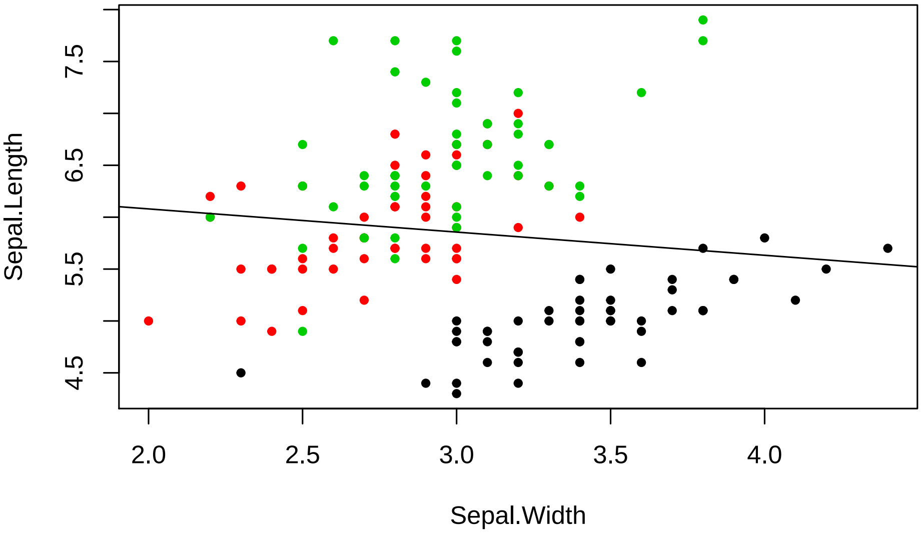
```r
resid(fit)
```

```
##           1           2           3           4           5           6
## -0.64445884 -0.95613937 -1.11146716 -1.23380326 -0.72212273 -0.25511441
##           7           8           9          10          11          12
## -1.16679494 -0.76679494 -1.47847547 -0.93380326 -0.29978662 -0.96679494
##          13          14          15          16          17          18
## -1.05613937 -1.55613937  0.16722169  0.15656612 -0.25511441 -0.64445884
##          19          20          21          22          23          24
##  0.02254948 -0.57745052 -0.36679494 -0.59978662 -1.12212273 -0.68913105
##          25          26          27          28          29          30
## -0.96679494 -0.85613937 -0.76679494 -0.54445884 -0.56679494 -1.11146716
##          31          32          33          34          35          36
## -1.03380326 -0.36679494 -0.41044220 -0.08810609 -0.93380326 -0.81146716
##          37          38          39          40          41          42
## -0.24445884 -0.82212273 -1.45613937 -0.66679494 -0.74445884 -1.51249211
##          43          44          45          46          47          48
## -1.41146716 -0.74445884 -0.57745052 -1.05613937 -0.57745052 -1.21146716
##          49          50          51          52          53          54
## -0.39978662 -0.78913105  1.18853284  0.58853284  1.06619674 -0.51249211
##          55          56          57          58          59          60
##  0.59918842 -0.20081158  0.51086895 -1.09015600  0.72152453 -0.72314769
##          61          62          63          64          65          66
## -1.07950043  0.04386063 -0.03482822  0.22152453 -0.27847547  0.86619674
##          67          68          69          70          71          72
## -0.25613937 -0.12314769  0.16517178 -0.36781990  0.08853284  0.19918842
##          73          74          75          76          77          78
##  0.33218010  0.19918842  0.52152453  0.74386063  0.89918842  0.84386063
##          79          80          81          82          83          84
##  0.12152453 -0.24548379 -0.49015600 -0.49015600 -0.12314769  0.07685231
##          85          86          87          88          89          90
## -0.45613937  0.23320506  0.86619674  0.28750789 -0.25613937 -0.46781990
##          91          92          93          94          95          96
## -0.44548379  0.24386063 -0.14548379 -1.01249211 -0.32314769 -0.15613937
##          97          98          99         100         101         102
## -0.17847547  0.32152453 -0.86781990 -0.20081158  0.51086895 -0.12314769
##         103         104         105         106         107         108
##  1.24386063  0.42152453  0.64386063  1.74386063 -1.06781990  1.42152453
##         109         110         111         112         113         114
##  0.73218010  1.47787727  0.68853284  0.47685231  0.94386063 -0.26781990
##         115         116         117         118         119         120
## -0.10081158  0.58853284  0.64386063  2.02254948  1.75451621 -0.03482822
##         121         122         123         124         125         126
##  1.08853284 -0.30081158  1.79918842  0.37685231  0.91086895  1.38853284
##         127         128         129         130         131         132
##  0.29918842  0.24386063  0.49918842  1.34386063  1.49918842  2.22254948
##         133         134         135         136         137         138
##  0.49918842  0.39918842  0.15451621  1.84386063  0.53320506  0.56619674
##         139         140         141         142         143         144
##  0.14386063  1.06619674  0.86619674  1.06619674 -0.12314769  0.98853284
##         145         146         147         148         149         150
##  0.91086895  0.84386063  0.33218010  0.64386063  0.43320506  0.04386063
```

The advantage of using the functions rather than accessing the components requested directly is that `coef()` and `resid()` are generic functions, and thus work with objects that are not `lm`-class objects (such as `glm`-class objects or objects provided in packages).

Seeing as we're working in two dimensions, plotting simple linear regression lines makes sense. The function `abline()` allows us to plot regression lines.

```r
plot(Sepal.Length ~ Sepal.Width, data = iris, col = Species, pch = 20)
abline(fit)
```

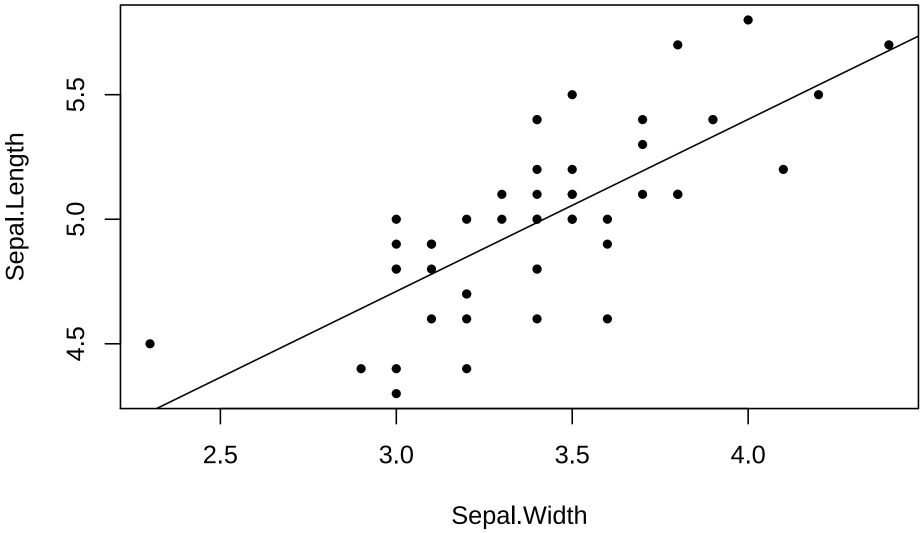


Do you notice something odd about this line? It's a downward-sloping line, but if we look at individual groups in the data, we see what appears to be a positive relationship between sepal length and width. This is a phenomenon known as **Simpson's paradox**, where the trend line in subgroups differs with the trend line across groups. In this case, setosa flower tend to have larger sepal widths but lower length than the other flowers, yet there's still a positive relationship between the two. Perhaps instead we should disaggregate and look at the regression line for only setosa flowers. Fortunately `lm()` provides an easy interface for selecting a subset of a data set.

```
(fit <- lm(Sepal.Length ~ Sepal.Width, data = iris,
                                       subset = Species == "setosa"))
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Sepal.Width, data = iris, subset = Species ==
##     "setosa")
##
## Coefficients:
## (Intercept)  Sepal.Width
##      2.6390       0.6905
```

```
plot(Sepal.Length ~ Sepal.Width, data = iris, subset = Species == "setosa",
     pch = 20)
abline(fit)
```



That line makes much more sense.

What if we want to call a function on our variables or apply some transformation to them? We can use function calls in formulas. For example, if we wanted to attempt to fit a log model to our data, we could do so like so:

```
lm(log(Sepal.Length) ~ Sepal.Width, data = iris, subset = Species == "setosa")
```

```
##
## Call:
## lm(formula = log(Sepal.Length) ~ Sepal.Width, data = iris, subset = Species ==
##     "setosa")
##
## Coefficients:
## (Intercept)  Sepal.Width
##      1.1368       0.1375
```

```
lm(Sepal.Length ~ log(Sepal.Width), data = iris, subset = Species == "setosa")
```

```
##
## Call:
## lm(formula = Sepal.Length ~ log(Sepal.Width), data = iris, subset = Species ==
##     "setosa")
##
## Coefficients:
##      (Intercept)  log(Sepal.Width)
##            2.202             2.287
```

However, for operations such as `^` or `*` that have special meaning in formulas, you should call such operations using `I()`, like so:

```
lm(Sepal.Length ~ I(Sepal.Width ^ 2), data = iris, subset = Species == "setosa")
```

```
##
## Call:
## lm(formula = Sepal.Length ~ I(Sepal.Width^2), data = iris, subset = Species ==
##     "setosa")
##
## Coefficients:
##      (Intercept)  I(Sepal.Width^2)
##           3.8113            0.1005
```

A handy function is the `scale()` function which centers the data around its mean and subtracts out the standard deviation.

```
# Showing what scale() does
with(iris, cbind(Sepal.Length, scale(Sepal.Width)))
```

```
##        Sepal.Length
##   [1,]          5.1  1.01560199
##   [2,]          4.9 -0.13153881
##   [3,]          4.7  0.32731751
##   [4,]          4.6  0.09788935
##   [5,]          5.0  1.24503015
##   [6,]          5.4  1.93331463
##   [7,]          4.6  0.78617383
##   [8,]          5.0  0.78617383
##   [9,]          4.4 -0.36096697
##  [10,]          4.9  0.09788935
##  [11,]          5.4  1.47445831
##  [12,]          4.8  0.78617383
##  [13,]          4.8 -0.13153881
##  [14,]          4.3 -0.13153881
##  [15,]          5.8  2.16274279
##  [16,]          5.7  3.08045544
##  [17,]          5.4  1.93331463
##  [18,]          5.1  1.01560199
##  [19,]          5.7  1.70388647
##  [20,]          5.1  1.70388647
##  [21,]          5.4  0.78617383
##  [22,]          5.1  1.47445831
##  [23,]          4.6  1.24503015
##  [24,]          5.1  0.55674567
##  [25,]          4.8  0.78617383
##  [26,]          5.0 -0.13153881
##  [27,]          5.0  0.78617383
##  [28,]          5.2  1.01560199
##  [29,]          5.2  0.78617383
##  [30,]          4.7  0.32731751
##  [31,]          4.8  0.09788935
##  [32,]          5.4  0.78617383
##  [33,]          5.2  2.39217095
##  [34,]          5.5  2.62159911
##  [35,]          4.9  0.09788935
##  [36,]          5.0  0.32731751
##  [37,]          5.5  1.01560199
##  [38,]          4.9  1.24503015
##  [39,]          4.4 -0.13153881
##  [40,]          5.1  0.78617383
##  [41,]          5.0  1.01560199
##  [42,]          4.5 -1.73753594
##  [43,]          4.4  0.32731751
##  [44,]          5.0  1.01560199
##  [45,]          5.1  1.70388647
##  [46,]          4.8 -0.13153881
##  [47,]          5.1  1.70388647
##  [48,]          4.6  0.32731751
##  [49,]          5.3  1.47445831
##  [50,]          5.0  0.55674567
##  [51,]          7.0  0.32731751
##  [52,]          6.4  0.32731751
##  [53,]          6.9  0.09788935
##  [54,]          5.5 -1.73753594
##  [55,]          6.5 -0.59039513
##  [56,]          5.7 -0.59039513
##  [57,]          6.3  0.55674567
##  [58,]          4.9 -1.50810778
##  [59,]          6.6 -0.36096697
##  [60,]          5.2 -0.81982329
##  [61,]          5.0 -2.42582042
##  [62,]          5.9 -0.13153881
##  [63,]          6.0 -1.96696410
##  [64,]          6.1 -0.36096697
##  [65,]          5.6 -0.36096697
##  [66,]          6.7  0.09788935
##  [67,]          5.6 -0.13153881
##  [68,]          5.8 -0.81982329
##  [69,]          6.2 -1.96696410
##  [70,]          5.6 -1.27867961
##  [71,]          5.9  0.32731751
##  [72,]          6.1 -0.59039513
##  [73,]          6.3 -1.27867961
##  [74,]          6.1 -0.59039513
##  [75,]          6.4 -0.36096697
##  [76,]          6.6 -0.13153881
##  [77,]          6.8 -0.59039513
##  [78,]          6.7 -0.13153881
##  [79,]          6.0 -0.36096697
##  [80,]          5.7 -1.04925145
##  [81,]          5.5 -1.50810778
##  [82,]          5.5 -1.50810778
##  [83,]          5.8 -0.81982329
##  [84,]          6.0 -0.81982329
##  [85,]          5.4 -0.13153881
##  [86,]          6.0  0.78617383
##  [87,]          6.7  0.09788935
##  [88,]          6.3 -1.73753594
##  [89,]          5.6 -0.13153881
##  [90,]          5.5 -1.27867961
##  [91,]          5.5 -1.04925145
##  [92,]          6.1 -0.13153881
##  [93,]          5.8 -1.04925145
##  [94,]          5.0 -1.73753594
##  [95,]          5.6 -0.81982329
##  [96,]          5.7 -0.13153881
##  [97,]          5.7 -0.36096697
##  [98,]          6.2 -0.36096697
##  [99,]          5.1 -1.27867961
## [100,]          5.7 -0.59039513
## [101,]          6.3  0.55674567
## [102,]          5.8 -0.81982329
## [103,]          7.1 -0.13153881
## [104,]          6.3 -0.36096697
## [105,]          6.5 -0.13153881
## [106,]          7.6 -0.13153881
## [107,]          4.9 -1.27867961
## [108,]          7.3 -0.36096697
## [109,]          6.7 -1.27867961
## [110,]          7.2  1.24503015
## [111,]          6.5  0.32731751
## [112,]          6.4 -0.81982329
## [113,]          6.8 -0.13153881
## [114,]          5.7 -1.27867961
## [115,]          5.8 -0.59039513
## [116,]          6.4  0.32731751
## [117,]          6.5 -0.13153881
## [118,]          7.7  1.70388647
## [119,]          7.7 -1.04925145
## [120,]          6.0 -1.96696410
## [121,]          6.9  0.32731751
## [122,]          5.6 -0.59039513
## [123,]          7.7 -0.59039513
## [124,]          6.3 -0.81982329
## [125,]          6.7  0.55674567
## [126,]          7.2  0.32731751
## [127,]          6.2 -0.59039513
## [128,]          6.1 -0.13153881
## [129,]          6.4 -0.59039513
## [130,]          7.2 -0.13153881
## [131,]          7.4 -0.59039513
## [132,]          7.9  1.70388647
## [133,]          6.4 -0.59039513
## [134,]          6.3 -0.59039513
## [135,]          6.1 -1.04925145
## [136,]          7.7 -0.13153881
## [137,]          6.3  0.78617383
## [138,]          6.4  0.09788935
## [139,]          6.0 -0.13153881
## [140,]          6.9  0.09788935
## [141,]          6.7  0.09788935
## [142,]          6.9  0.09788935
## [143,]          5.8 -0.81982329
## [144,]          6.8  0.32731751
## [145,]          6.7  0.55674567
## [146,]          6.7 -0.13153881
## [147,]          6.3 -1.27867961
## [148,]          6.5 -0.13153881
## [149,]          6.2  0.78617383
## [150,]          5.9 -0.13153881
```

```r
lm(scale(Sepal.Length) ~ scale(Sepal.Width), data = iris,
   subset = Species == "setosa")
```

```
##
## Call:
## lm(formula = scale(Sepal.Length) ~ scale(Sepal.Width), data = iris,
##     subset = Species == "setosa")
##
## Coefficients:
##        (Intercept)  scale(Sepal.Width)
##            -1.3203              0.3635
```

Note that the interpretation of the coefficients when computed on scaled data differs from the interpretation for unscaled data.

Above a line was plotted. This line represents the predicted sepal lengths given the sepal widths. It's possible to get predicted values at specific points on the line, for specific sepal widths. The function `predict()` (a generic function) gets predicted values. To use it, one must give a data frame resembling the one from which the model was estimated, except perhaps leaving out the variable we're attempting to predict.

```
predict(fit)  # Returns predicted values at observed sepal widths
```

```
##        1        2        3        4        5        6        7        8
## 5.055715 4.710470 4.848568 4.779519 5.124764 5.331911 4.986666 4.986666
##        9       10       11       12       13       14       15       16
## 4.641421 4.779519 5.193813 4.986666 4.710470 4.710470 5.400960 5.677156
##       17       18       19       20       21       22       23       24
## 5.331911 5.055715 5.262862 5.262862 4.986666 5.193813 5.124764 4.917617
##       25       26       27       28       29       30       31       32
## 4.986666 4.710470 4.986666 5.055715 4.986666 4.848568 4.779519 4.986666
##       33       34       35       36       37       38       39       40
## 5.470009 5.539058 4.779519 4.848568 5.055715 5.124764 4.710470 4.986666
##       41       42       43       44       45       46       47       48
## 5.055715 4.227128 4.848568 5.055715 5.262862 4.710470 5.262862 4.848568
##       49       50
## 5.193813 4.917617
```

```
predict(fit, newdata = data.frame(Sepal.Width = c(2, 3, 4)))
```

```
##        1        2        3
## 4.019981 4.710470 5.400960
```

When making predictions, though, you should avoid **extrapolation**, which is making predictions outside of the range of the data. Going slightly outside the range of the data is fine, but well outside the range is a problem. We will talk later about estimating prediction errors, but leaving the range of the data tends to produce predictions with high error. More importantly, though, while a linear model may be appropriate in a certain range of the data, the same linear model may not be appropriate outside of the range; data might be *locally* linear rather than *globally* linear. This means that outside of the range the model is likely to be wrong, and a different model should be used. Since no data was observed in that range, though, determining and estimating that model is hard.

Lecture 9

Simple Linear Regression Inference and Diagnostics

Linear regression itself does not require statistics to be done and one needs few assumptions for the *ability* to estimate a regression model. Statistics steps in when one wants to study the properties of a regression model. Using statistics, we can make statements about what the value of the *population* parameters are, decide between competing models, express uncertainty in mean values and predictions, and more. We will see techniques as well for assessing the quality of a model.

For reference, here's the regression model we are considering:

$$y_i = \beta_0 + \beta_1 x_{1i} + \cdots + \beta_k x_{ki} + \epsilon_i.$$

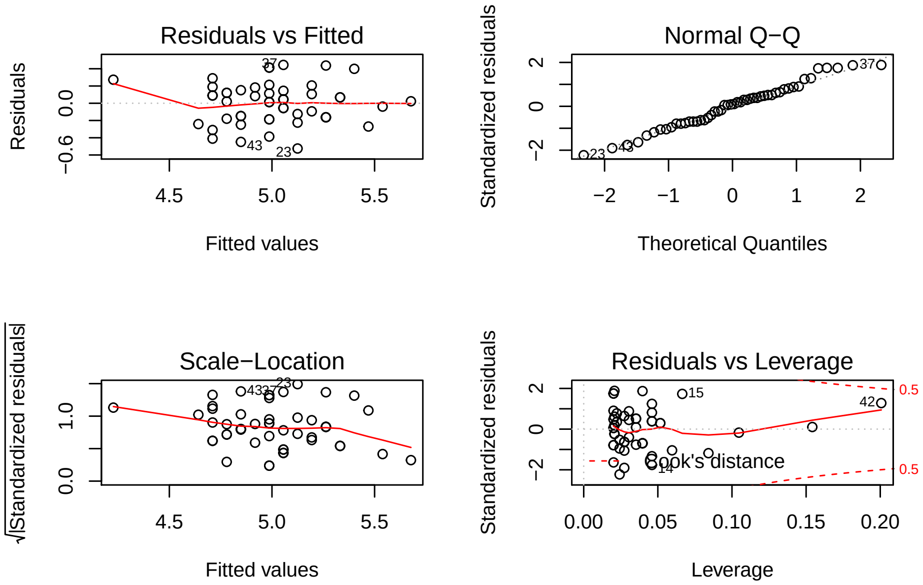
Residual Properties

Linear models fitted via `lm()` can be plotted using `plot()`. The result is a sequence of four plots useful for assessing the quality of a linear model fit. This is in addition to a basic plot visualizing the linear model with the data.

```r
(fit <- lm(Sepal.Length ~ Sepal.Width, data = iris,
           subset = Species == "setosa"))
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Sepal.Width, data = iris, subset = Species ==
##     "setosa")
##
## Coefficients:
## (Intercept)  Sepal.Width
##      2.6390       0.6905
```

```r
old_par <- par()
par(mfrow = c(2, 2))
plot(fit)
```



```r
par(old_par)
```

```
## Warning in par(old_par): graphical parameter "cin" cannot be set

## Warning in par(old_par): graphical parameter "cra" cannot be set

## Warning in par(old_par): graphical parameter "csi" cannot be set

## Warning in par(old_par): graphical parameter "cxy" cannot be set

## Warning in par(old_par): graphical parameter "din" cannot be set

## Warning in par(old_par): graphical parameter "page" cannot be set
```

The meaning of these plots, from left to right and top to bottom:

1. We plot the estimated residuals of the model against the predicted value of the observations in the data set, according to the model. What we want to see is a “cloudy” shape, with no particularly strong patterns. Patterns could indicate an inappropriate model form; a simple linear model may not be appropriate and perhaps a different functional form is needed. We would also want to see constant “spread” in the data. If it seems that data is more spread out for some predicted values than others, this could indicate a phenomenon known as **heteroskedasticity** in the data (where the variance of the residuals depend on the data). We want the red line to be roughly straight. My own opinion is that for the plot above, the line is

straight enough and there's no strong evidence of heteroskedasticity (that is, the data is **homoskedastic**).

2. We create a Q-Q plot for the residuals of the data to see if they appear to be Normally distributed. Statistical procedures often assume Normally distributed residuals. We may be able to proceed without Normally distributed residuals, but some common procedures (such as prediction intervals) may fail.
3. This is a scale-location plot, which compares predicted values to the square root of the absolute value of the residuals. This plot is useful for identifying heteroskedasticity in the model. We want a "cloudy" shape with a roughly straight and flat red line. That appears to be the case here.
4. Least-squares regression ties intimately to the concept of mean and standard deviation, and as you should know from previous studies, means and standard deviations are affected by strong outliers. The notion of "outlier" is murkier in the regression context, yet still we care about "influential points". The final plot compares the residuals to their leverage. In short, what we are looking for are points that fall outside a quantity known as Cook's distance, which manifests as a point in one of the corners on the right-hand side of the plot outside of the Cook's distance bands. Such points seem to have a strong influence on the resulting regression line and removing them could result in a regression line quite different from before. In this case, while one observation gets close to the Cook's distance band, it doesn't cross, and thus it doesn't appear that there are strong outliers "biasing" our regression estimates.

(The above plots actually work with standardized residuals, or the residuals scaled to have variance 1.)

Compare the above model to a model estimated for home runs against at bats in the `batting` (**UsingR**) data set.

```r
library(UsingR)
```

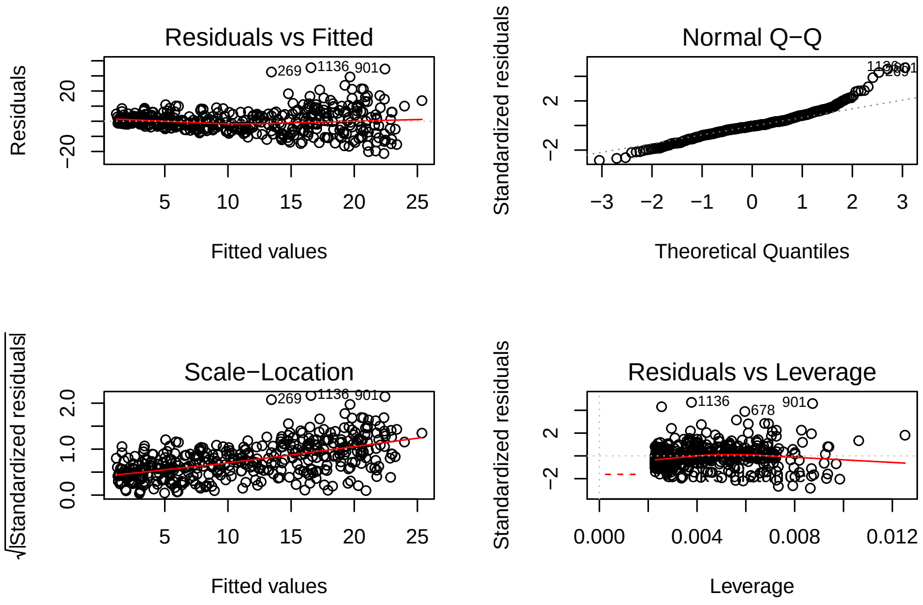
```
## Loading required package: HistData

## Loading required package: Hmisc

## Loading required package: lattice

## Loading required package: survival

## Loading required package: Formula

## Loading required package: ggplot2

##
## Attaching package: 'Hmisc'

## The following objects are masked from 'package:base':
##
##     format.pval, units
```

```
##
## Attaching package: 'UsingR'
## The following object is masked from 'package:survival':
##
##     cancer
```

```
(fit2 <- lm(HR ~ AB, data = batting))
```

```
##
## Call:
## lm(formula = HR ~ AB, data = batting)
##
## Coefficients:
## (Intercept)           AB
##    -2.94675      0.04067
```

```
old_par <- par()
par(mfrow = c(2, 2))
plot(fit2)
```



```
par(old_par)
```

```
## Warning in par(old_par): graphical parameter "cin" cannot be set
## Warning in par(old_par): graphical parameter "cra" cannot be set
## Warning in par(old_par): graphical parameter "csi" cannot be set
## Warning in par(old_par): graphical parameter "cxy" cannot be set
## Warning in par(old_par): graphical parameter "din" cannot be set
```

```
## Warning in par(old_par): graphical parameter "page" cannot be set
```

Model Inference

Given an `lm()` fit called `fit`, we can see tables containing statistical results for our models using `stargazer(fit)` or `summary(fit)`.

```r
(inf <- summary(fit))  # Saving summary results; could be useful
```

```
## 
## Call:
## lm(formula = Sepal.Length ~ Sepal.Width, data = iris, subset = Species ==
##     "setosa")
## 
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -0.52476 -0.16286  0.02166  0.13833  0.44428 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept)   2.6390     0.3100   8.513 3.74e-11 ***
## Sepal.Width   0.6905     0.0899   7.681 6.71e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 0.2385 on 48 degrees of freedom
## Multiple R-squared:  0.5514, Adjusted R-squared:  0.542 
## F-statistic: 58.99 on 1 and 48 DF,  p-value: 6.71e-10
```

```r
str(inf)
```

```
## List of 11
##  $ call         : language lm(formula = Sepal.Length ~ Sepal.Width, data = iris, subset = Spec
##  $ terms        :Classes 'terms', 'formula'  language Sepal.Length ~ Sepal.Width
##   .. ..- attr(*, "variables")= language list(Sepal.Length, Sepal.Width)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "Sepal.Length" "Sepal.Width"
##   .. .. .. ..$ : chr "Sepal.Width"
##   .. ..- attr(*, "term.labels")= chr "Sepal.Width"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv> 
##   .. ..- attr(*, "predvars")= language list(Sepal.Length, Sepal.Width)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "Sepal.Length" "Sepal.Width"
##  $ residuals    : Named num [1:50] 0.0443 0.1895 -0.1486 -0.1795 -0.1248 ...
##   ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
##  $ coefficients : num [1:2, 1:4] 2.639 0.6905 0.31 0.0899 8.5125 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:2] "(Intercept)" "Sepal.Width"
##   .. ..$ : chr [1:4] "Estimate" "Std. Error" "t value" "Pr(>|t|)"
##  $ aliased      : Named logi [1:2] FALSE FALSE
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "Sepal.Width"
##  $ sigma        : num 0.239
##  $ df           : int [1:3] 2 48 2
##  $ r.squared    : num 0.551
##  $ adj.r.squared: num 0.542
##  $ fstatistic   : Named num [1:3] 59 1 48
##   ..- attr(*, "names")= chr [1:3] "value" "numdf" "dendf"
##  $ cov.unscaled : num [1:2, 1:2] 1.689 -0.487 -0.487 0.142
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:2] "(Intercept)" "Sepal.Width"
##   .. ..$ : chr [1:2] "(Intercept)" "Sepal.Width"
##  - attr(*, "class")= chr "summary.lm"
```

```r
library(stargazer)
stargazer(fit, type = "text")
```

```
##
## ===============================================
##                         Dependent variable:
##                     ---------------------------
##                            Sepal.Length
## -----------------------------------------------
## Sepal.Width                  0.690***
##                               (0.090)
##
## Constant                     2.639***
##                               (0.310)
##
## -----------------------------------------------
## Observations                    50
## R2                             0.551
## Adjusted R2                    0.542
## Residual Std. Error       0.239 (df = 48)
## F Statistic           58.994*** (df = 1; 48)
## ===============================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

Let’s explain these tables.

Coefficient Standard Errors

The estimated regression coefficients are statistics estimated from a sample, so like other statistics you've seen (sample mean or sample proportion), they have a standard error. The estimated standard error is given below:

```
vcov(fit)
```

```
##              (Intercept)  Sepal.Width
## (Intercept)   0.09610887 -0.027704441
## Sepal.Width  -0.02770444  0.008081809
```

Actually I lied. When discussing regression models the notion of "standard error" still exists but needs to change. See, we're not estimating just one statistic, but two: a slope and an intercept term. As a result, we need to account not just for how the two statistics themselves vary but also how the statistics vary *jointly*. That is, we need to account for correlation in the sample estimates.

The above matrix produced by the function `vcov()` is a **covariance matrix**, where the entries on the diagonal of the matrix are variances of random variables and the off-diagonal entries are the covariances between respective random variables. (Since the covariance is a symmetric function, all covariance matrices are symmetric. Additionally, since variances are non-negative, the diagonal of a covariance matrix is non-negative. In fact there are many properties of covariance matrices that are ultimately linear algebra discussions we won't have here.) in this case, the diagonal of the matrix produced by `vcov()` contains squared standard errors of the coefficients while the off-diagonal entries are the covariances between those parameter estimates. If we want correlations we can call `cov2cor()` on the covariance matrix.

```
cov2cor(vcov(fit))
```

```
##              (Intercept) Sepal.Width
## (Intercept)    1.0000000  -0.9940617
## Sepal.Width   -0.9940617   1.0000000
```

For this correlation matrix, the diagonal entries are all 1 since random variables are perfectly correlated with themselves. The interesting part is the off-diagonal entries; here we see extremely strong negative correlations. Our interpretation of this is that the intercept and the slope term of the regression line vary in opposite directions; if the slope term is too small, the intercept term probably is too large.

That said, if we want the standard errors of the coefficients without considering their covariation, we can extract the diagonal of the covariance matrix using the function `diag()` (which is also the function used for creating diagonal matrices from vectors in R). Then we can take the square root of the result to get the standard errors of the coefficients.

```
coef_stderr <- function(...) {
  sqrt(diag(vcov(...)))
}

coef_stderr(fit)
```

```
## (Intercept) Sepal.Width
##  0.31001431  0.08989888
```

Compare the above to the output of `summary(fit)`.

t-Tests

Consider the following matrix:

```
inf$coefficients
```

```
##              Estimate Std. Error  t value     Pr(>|t|)
## (Intercept) 2.6390012 0.31001431 8.512514 3.742438e-11
## Sepal.Width 0.6904897 0.08989888 7.680738 6.709843e-10
```

Notice the last two columns. We can perform hypothesis testing to determine the value of the population coefficients corresponding with our estimates. That is, we can decide between:

$$H_0 : \beta_j = \beta_{j0}$$

and

$$H_A : \begin{cases} \beta_j < \beta_{j0} \\ \beta_j \neq \beta_{j0} \\ \beta_j > \beta_{j0} \end{cases}$$

Assume that the standard errors ϵ_i in the regression model are *i.i.d.* and Normally distributed with mean zero. Then we can decide between H_0 and H_A by using the t-statistic:

$$t = \frac{\hat{\beta}_j - \beta_{k0}}{SE(\hat{\beta}_j)}$$

where $SE(\hat{\beta}_j)$ is the standard error of the statistic $\hat{\beta}_j$. Under the null hypothesis, the t-statistic follows a $t(n-k-1)$ distribution; notice that the degrees of freedom of the statistic is $n-k-1$, *not* $n-1$. Generally in statistics, the degrees of freedom are n minus the number of parameters being estimated. In the simple mean testing context, one parameter was estimated: the mean, μ. In fact, we can view hypothesis testing in that context as inference for the mean model:

$$y_i = \mu + \epsilon_i.$$

In linear regression, there are $k + 1$ parameters being estimated: the k slope terms plus the intercept term. (If suppressing the constant, there would be k parameters estimated, so the degrees of freedom would be $n - k$.)

Otherwise hypothesis tests regarding the value of regression coefficients are exactly like the t-tests you've seen before. `summary.lm()` reports the results for hypothesis tests checking whether the coefficients are zero or not. (That is, the tests are two-sided and $\beta_{j0} = 0$.) This is the default hypothesis test in linear regression models since it translates to whether an explanatory variables has a linear effect on a response variable or not. If not, it may be inappropriate to include the variable in the model.

The regression table returned by `stargazer()` more closely resembles regression tables commonly seen in papers and books, with standard errors reported below coefficient estimates and stars appearing beside the coefficients. More stars appear for increasingly small p-values, and the cutoffs for the number of stars shown should be listed nearby. (Notice that `summary.lm()` also uses stars but at a different scale than that given by `stargazer()`.)

Of course, we can do our own hypothesis tests for what values the coefficients should be. First let's compute t-statistics:

```r
coef_tstat <- function(fit, beta0 = 0) {
  se <- coef_stderr(fit)
  beta <- coef(fit)
  (beta - beta0) / se
}

coef_tstat(fit)
```

```
## (Intercept) Sepal.Width
##    8.512514    7.680738
```

```r
coef_tstat(fit, beta0 = c(2, 1))
```

```
## (Intercept) Sepal.Width
##    2.061199   -3.442871
```

Then compute degrees of freedom.

```r
deg_free <- function(fit) {
  n <- length(resid(fit))
  k <- length(coef(fit))
  n - k
}

deg_free(fit)
```

```
## [1] 48
```

Then we can test a one-side alternative hypothesis (in thise case, $H_A : \beta_0 > 2$) like so:

```
pt(coef_tstat(fit, beta0 = 2)[1], df = deg_free(fit), lower.tail = FALSE)
```

```
## (Intercept)
##  0.02236164
```

Confidence Intervals

In mean models we compute confidence intervals for the mean. Naturally we would like to compute confidence intervals for the population model coefficients as well. However, since we're estimating multiple parameters, the conversation isn't quite the same.

On the one hand we can consider an interval for each β_j without concerning ourselves with the values of the other coefficients. If we wanted a confidence interval for *just* β_j, we could compute one using

$$\hat{\beta}_j \pm t^* SE(\hat{\beta}_j)$$

with t^* be the corresponding critical value from a t distribution with $n - k - 1$ degrees of freedom.

The code below obtains confidence intervals for each of the parameters in the model in this way.

```
coef_conf_int <- function(fit, conf.level = 0.95) {
  beta <- coef(fit)
  se <- coef_stderr(fit)
  nu <- deg_free(fit)
  tstar <- qt((1 - conf.level) / 2, df = nu, lower.tail = FALSE)
  moe <- se * tstar
  cbind(lower = beta - moe, upper = beta + moe)
}

coef_conf_int(fit)
```

```
##                 lower     upper
## (Intercept) 2.0156757 3.2623268
## Sepal.Width 0.5097359 0.8712435
```

However, these are *marginal* confidence intervals that are valid when considering the model parameters *separately*. The *join* confidence interval (where we hope that *both* β_0 *and* β_1 fall into the region) is quite different. For starters, a confidence "interval" doesn't make sense when talking about something that is essentially two-dimensional: the vector $(\beta_0, \beta_1)^\top$. Instead we need to talk about a confidence *region*, a set (this time in a two-dimensional plane) that hopefully contains the values of the population parameters. (In general, these sets are subsets of $k+1$ dimensional space; remember that k is the number of parameters in the model excluding the intercept and residual standard deviation.) If we were to use just the marginal confidence intervals found above, the resulting region would be a square/cube/hypercube, but the probability the resulting region would capture the parameter values *would not be the confidence level*. Thus such a region is inappropriate.

Statisticians often consider a **confidence ellipse** (in high dimensions this would be a hyperellipse, but we'll still call it an ellipse). A confidence ellipse accounts for the correlation in the sample estimates, and any point in the ellipse represents a plausible combination of values for the true parameter coefficients.

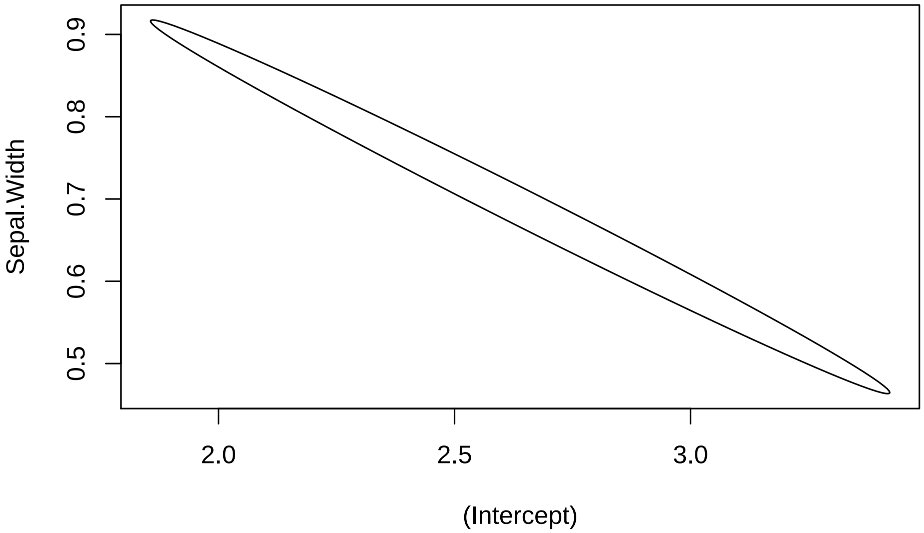
The **ellipse** package allows for visualizing confidence ellipses. Below I show how it can be used for visualizing a confidence ellipse.

```r
library(ellipse)
```

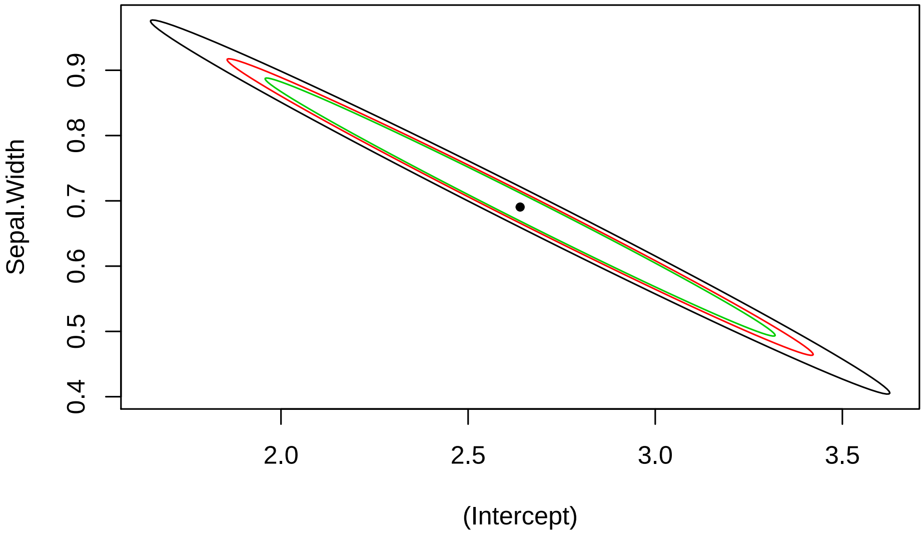
```
##
## Attaching package: 'ellipse'
```

```
## The following object is masked from 'package:graphics':
##
##     pairs
```

```r
plot_conf_ellipse <- function(..., col = 1, add = FALSE) {
  if (add) {
    pfunc = lines
  } else {
    pfunc = plot
  }
  pfunc(ellipse(...), type = "l", col = col)
}

plot_conf_ellipse(fit)
```



```
plot_conf_ellipse(fit, level = 0.99)
plot_conf_ellipse(fit, level = 0.95, add = TRUE, col = 2)
plot_conf_ellipse(fit, level = 0.90, add = TRUE, col = 3)
points(coef(fit)[1], coef(fit)[2], pch = 20)
```



Our takeaway from the above plots is that the coefficient estimates for the intercept and slope terms are highly correlated, and if the slope is too small, the intercept is likely to be too large.

R^2

The R^2 value, also known as the **coefficient of determination**, is a measure of how well the regression line fits the data. If $SSE = \sum_{i=1}^{n}(y_i - \bar{y})^2$ and $SSR = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2$, then

$$R^2 = 1 - \frac{SSR}{SSE}.$$

The R^2 value can be interpreted as the percentage of variation in the response variable that can be explained by variation in the explanatory variable. In fact, it turns out that in this simple linear regression context, $R^2 = r^2$, where r is the correlation coefficient. In this way one can generalize the notion of correlation to more than one variable (the other way is the covariance matrix).

We can extract R^2 from the summary of a linear model like so:

```r
inf$r.squared
```

```
## [1] 0.5513756
```

Despite the interpretation presented above, don't get too excited about interpreting R^2. Just because R^2 is high doesn't mean there is a causal relationship between the variables; it just means there's an association between them.

Adjusted R^2 is interpreted like R^2 but accounts for the number of parameters in the model. It turns out that R^2 cannot decrease as one adds more variables, so one could increase R^2 by adding irrelevant variables to the model. (The resulting model would be considered **overfitted**, where errors in the estimated model are small but the model behaves poorly out of sample.) Adjusted R^2 is a penalized version of R^2 and equals $\bar{R}^2 = 1 - (1 - R^2)\frac{n-1}{n-k-1}$.

We can extract adjusted R^2 using:

```r
inf$adj.r.squared
```

```
## [1] 0.5420292
```

In the case of simple linear regression the distinction between the two metrics matters little, but for more complex models we almost always consider adjusted R^2 rather than R^2.

F-Test

The F test can be used to determine if additional coefficients in a regression model in some sense "add value" to the model; more specifically, it checks if all additional coefficients are zero or not. Here we would be using the F-test to decide if the simple linear regression model is better than the mean model.

The F statistic is used for inference and can be extracted from the summary of a linear model like so:

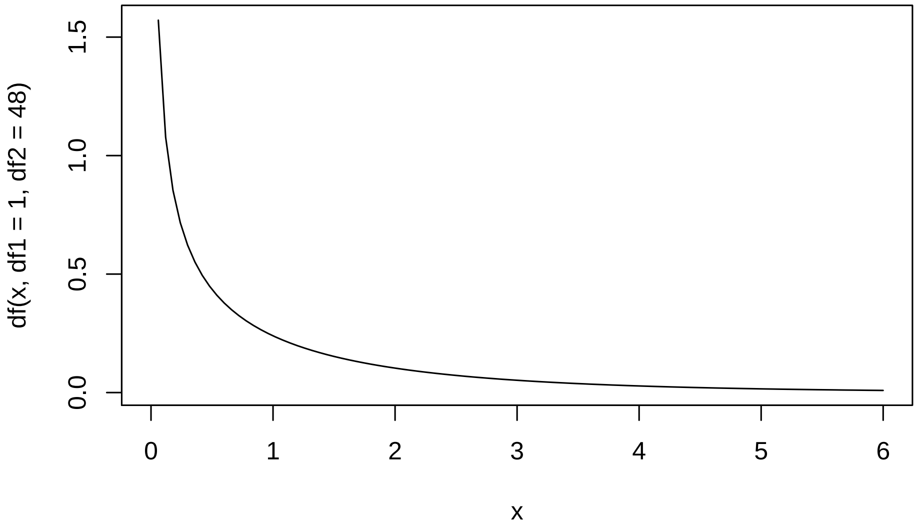
```
inf$fstatistic
```

```
##    value    numdf    dendf
## 58.99373  1.00000 48.00000
```

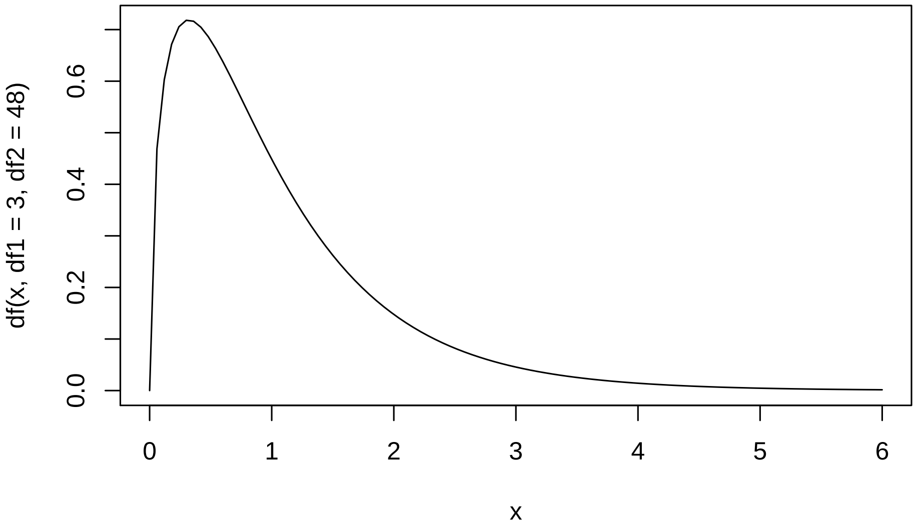
Inference with the F statistic is done using the f distribution, which depends on two parameters: the numerator degrees of freedom and the denominator degrees of freedom. If X is an f-distributed random variable, we say $X \sim f(\nu_{\text{num}}, \nu_{\text{den}})$, where ν_{num} and ν_{den} are the numerator and denominator degrees of freedom, respectively. In this case, if we denote the observed f statistic with $\hat{f}$, the p-value is computed via $P(X > \hat{f})$.

Here is a visualization of the F distribution.

```
curve(df(x, df1 = 1, df2 = 48), from = 0, to = 6)
```



```
curve(df(x, df1 = 3, df2 = 48), from = 0, to = 6)
```



Below is a function that computes the p-value for the F test:

```
ftest_pval <- function(fit) {
  f <- summary(fit)$fstatistic
  pf(f[["value"]], df1 = f[["numdf"]], df2 = f[["dendf"]], lower.tail = FALSE)
}

ftest_pval(fit)
```

```
## [1] 6.709843e-10
```

The p-value for this model is very small, suggesting that the explanatory variable belongs in our linear model for the response variable. (Well, whether it "belongs" is a strong statement; there could be a better model not including this variable.) For now the F-test doesn't seem to offer any advantage over tests for just the regression coefficients, but one should note that the two are not the same and one is not a substitute for the other. Particularly, for models with multiple explanatory variables, just because none of the explanatory variables' regression coefficients are statistically different from zero doesn't mean that the mean model is better than the regression model. On the flip side, a single statistically significant coefficient in a large linear model could be a false positive; the result of the F test would warn of this possibility. The F test again represents an overall test for statistical significance while the t-tests would be individual tests helping decide which coefficients are not zero.

The f-test can also be used to decide between a regression model and another model with more explanatory variables. We will revisit such a test later.

Point-Wise Inference

Our inference procedures so far have focused on the properties of the *model*, but we can also have discussions about the value of the model at particular points, for select values of the explanatory variables. If Y represents the response variable understood as a random variable, we are studying the behavior of $Y \mid |X = x$, where X is the explanatory variable viewed as a random variable. We would like to address two particular questions:

- What is the expected value of Y given $X = x$? That is, what is the conditional mean of Y when $X = x$? We will answer this with a point-wise confidence interval.
- If we know $X = x$, what are plausible values of Y? That is, we would like to predict the value of Y given $X = x$; this is a point-wise prediction interval.

Again we will assume that the residuals are Normally distributed. If the sample size is large this won't matter for point-wise confidence intervals, but this assumption matters for prediction intervals at any sample size.

The function `predict()` when called on an `lm`-class object gives estimated conditional means for regression models, but it also does more. It can also give confidence and prediction intervals. Let's write a function returning a closure giving an easy interface to `predict()` for a given data set.

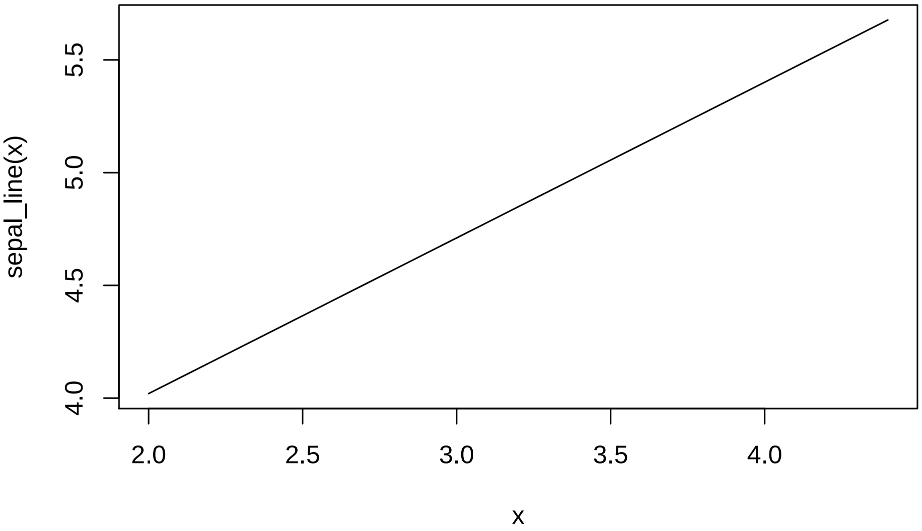
```
predict_func <- function(fit) {
  response <- names(fit$model)[[1]]
  explanatory <- names(fit$model)[[2]]
  function(x, ...) {
    dat <- data.frame(x)
    names(dat) <- explanatory
    predict(fit, dat, ...)
  }
}
```

Let's demonstrate the closure on the `iris` model.

```
sepal_line <- predict_func(fit)
sepal_line(2:4)
```

```
##        1        2        3
## 4.019981 4.710470 5.400960
```

```
with(iris, {
  curve(sepal_line, from = min(Sepal.Width), to = max(Sepal.Width))
})
```



Now note that the arguments being passed to `sepal_line()` other than the first argument `x` are actually arguments for `predict()`. We just made a nicer interface for `predict()`. Anyway, `predict()` is able to compute both point-wise confidence and prediction intervals. We do so by setting the `interval` argument to either `"confidence"` or `"prediction"`, depending on our desires. By default these are 95% intervals, but we can change the level by setting the `level` argument.

```
sepal_line(2:4, interval = "confidence", level = 0.99)  # Point-wise CIs
```

```
##        fit      lwr      upr
## 1 4.019981 3.663961 4.376001
## 2 4.710470 4.573218 4.847722
## 3 5.400960 5.236004 5.565916
```

```
sepal_line(2:4, interval = "prediction", level = 0.99)  # Point-wise PIs
```

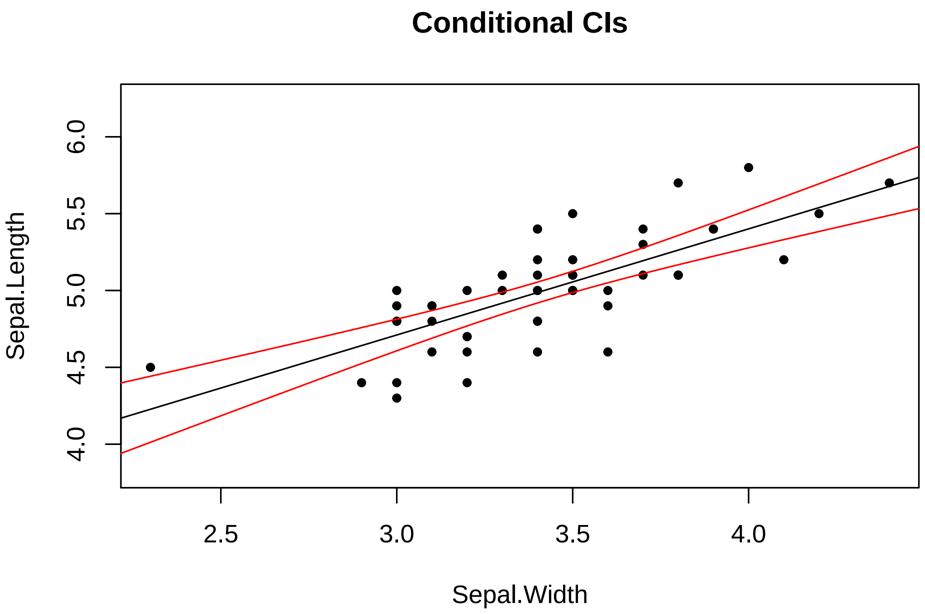
```
##        fit      lwr      upr
## 1 4.019981 3.287780 4.752182
## 2 4.710470 4.056096 5.364845
## 3 5.400960 4.740219 6.061701
```

Note that what's returned is a matrix with three columns; the first is a column with the value of the regression line at that point, the second column the lower bound of the interval, the third the upper bound.

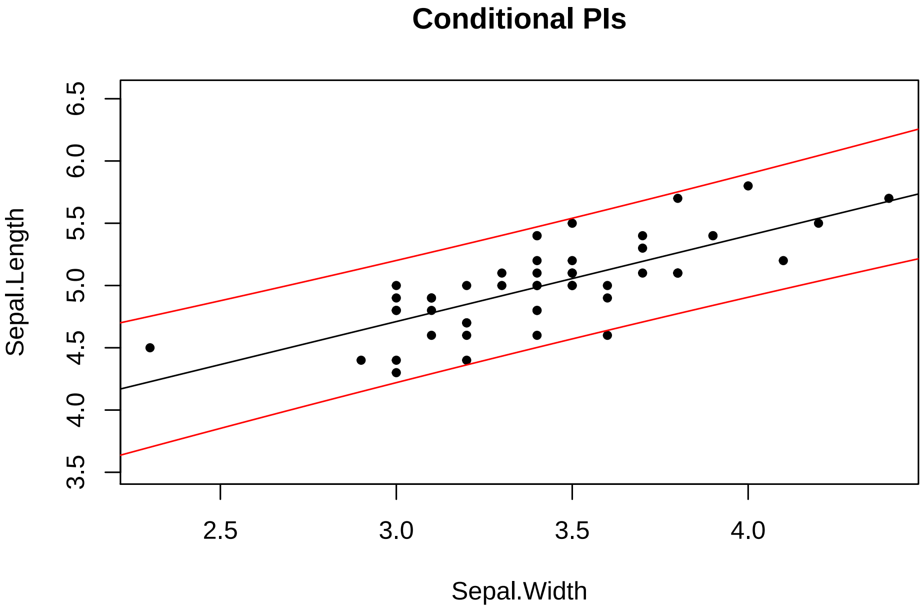
We would like to visualize these intervals. The function below will plot these intervals along with the original data.

```
plot_fit <- function(fit, interval = "confidence", level = 0.95, len = 1000,
                     ...) {
```

```
  fit_func <- predict_func(fit)
  x_range <- c(min(fit$model[[2]]), max(fit$model[[2]]))
  x_vals <- seq(from = x_range[[1]] * 0.9, to = x_range[[2]] * 1.1,
                length = len)
  interval_mat <- fit_func(x_vals, interval = interval, level = level)
  y_range <- c(min(c(interval_mat[, 2], fit$model[[1]])),
               max(c(interval_mat[, 3], fit$model[[1]])))
  plot(fit$model[2:1], xlim = x_range, ylim = y_range, ...)
  abline(fit)
  lines(x_vals, interval_mat[, 2], col = "red")
  lines(x_vals, interval_mat[, 3], col = "red")
}

plot_fit(fit, main = "Conditional CIs", pch = 20)
```



```
plot_fit(fit, main = "Conditional PIs", pch = 20, interval = "prediction")
```



Notice two things: one, the intervals are curved. The uncertainty of our estimates increases as we approach the ends of the range of the explanatory variable. Two, prediction intervals are wider than confidence intervals. Due to the nature of what the two intervals try to describe, this should not be surprising.

When using these intervals, one should avoid **extrapolation**: using the intervals to reach conclusions about the properties of the response interval well outside of the observed range of the explanatory variable. Extrapolation is problematic for two reasons: the standard errors of the statistics increase outside of this range, perhaps becoming quite large; and the likelihood a linear model is appropriate outside of this range becomes more uncertain. Regarding the latter point, it's possible that within some range of the explanatory variable the relationship between the explanatory and response variables is roughly linear, while outside this interval other forces, such as physical limitations, renders a linear model inappropriate even as an approximation for the truth. While one may wander slightly outside of the range of the explanatory variable, they should not wander far, and with great caution.

Lecture 10

Multiple Linear Regression

My presentation up to now has been focused on the simple linear model but has suggested the possibility of multiple explanatory variables. In fact regression with multiple variables is technically not much different from simple linear regression. Much of what we've said transfers over, though the issues of model selection become more interesting.

Estimation

Estimating a model with multiple explanatory variables is easy; just add those variables into the model.

Let's for example study the data set `mtcars` and examine the influence of variables on the miles per gallon of a vehicle.

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
fit <- lm(mpg ~ disp + hp, data = mtcars)
summary(fit)
```

```
##
## Call:
## lm(formula = mpg ~ disp + hp, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.7945 -2.3036 -0.8246  1.8582  6.9363 
## 
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 30.735904   1.331566  23.083  < 2e-16 ***
## disp        -0.030346   0.007405  -4.098 0.000306 ***
## hp          -0.024840   0.013385  -1.856 0.073679 .  
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 3.127 on 29 degrees of freedom
## Multiple R-squared:  0.7482, Adjusted R-squared:  0.7309 
## F-statistic: 43.09 on 2 and 29 DF,  p-value: 2.062e-09
```

```r
library(stargazer)
stargazer(fit, type = "text")
```

```
## 
## ===============================================
##                         Dependent variable:    
##                     ---------------------------
##                                 mpg            
## -----------------------------------------------
## disp                         -0.030***         
##                               (0.007)          
##                                                
## hp                            -0.025*          
##                               (0.013)          
##                                                
## Constant                     30.736***         
##                               (1.332)          
##                                                
## -----------------------------------------------
## Observations                    32             
## R2                             0.748           
## Adjusted R2                    0.731           
## Residual Std. Error       3.127 (df = 29)      
## F Statistic           43.095*** (df = 2; 29)   
## ===============================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

In statistics, a variable that takes the values of either 0 or 1 depending on whether a condition is true or not is called a **dummy variable** or **indicator variable**. In the `mtcars` data set, the variables `vs` and `am` are indicator variables tracking the shape of the engine of a car and whether a car has an automatic transmission, respectively. These variables are already conveniently encoded as 0/1 variables.

```
fit2 <- lm(mpg ~ disp + hp + am, data = mtcars)
stargazer(fit2, type = "text")
```

```
##
## ===============================================
##                         Dependent variable:
##                     ---------------------------
##                                 mpg
## -----------------------------------------------
## disp                          -0.014
##                               (0.009)
##
## hp                           -0.041***
##                               (0.014)
##
## am                            3.796**
##                               (1.424)
##
## Constant                     27.866***
##                               (1.620)
##
## -----------------------------------------------
## Observations                    32
## R2                             0.799
## Adjusted R2                    0.778
## Residual Std. Error       2.842 (df = 28)
## F Statistic           37.149*** (df = 3; 28)
## ===============================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

Oh, look at that! Our original model saved in `fit` had a statistically significant coefficient for `disp` but that's not the case for `fit2`. We will need to look into this. It's possible that the reason why this occured is because the original model suffered from **omitted variable bias (OMB)**, where two potential regressors were correlated with each other but only one of them had a meaningful effect on the response variable. But perhaps the issue instead is that the new model suffers from **multicollinearity**, which means that there is a strong linear relationship between two variables; in the case of **perfect multicollinearity**, where one regressor is *exactly* a linear combination of the other variables (that is, we could say that some variables $z_1, \ldots, z_p$ satisfy $z_1 = b_1 + b_2 z_2 + \ldots + b_p z_p$ for some coefficients $b_1, \ldots, b_p$), the model cannot be estimated at all (though often software detects this and simply removes variables until the perfect multicollinearity is gone). A consequence of multicollinearity is large standard errors in model coefficients, which makes precise estimation and inference difficult. One can understand the phenomenon as struggling to distinguish the effects of two variables that are strongly related to each other. A third possibility is that one of these variables are **irrelevant** and increase the standard errors of the other coefficient estimates in the model. Here either omitted variable bias or multicollinearity are likely producing these results, and we would need to decide between them. (I'm inclined to believe the issue is OMB: `am` was an omitted variable and `disp` actually might not belong in the model.)

Suppose we want to account for the cylinders in our model, along with the shape of the engine. We could include `cyl` and `vs` as variables. Additionally, we could allow for an interaction between the variables by multiplying `cyl` with `vs`. The resulting model is estimated below:

```r
fit3 <- lm(mpg ~ disp + hp + am + cyl * vs, data = mtcars)
stargazer(fit3, type = "text")
```

```
## 
## ===============================================
##                         Dependent variable:    
##                     ---------------------------
##                                 mpg            
## -----------------------------------------------
## disp                          -0.010           
##                               (0.011)          
##                                                
## hp                           -0.038**          
##                               (0.015)          
##                                                
## am                            3.867**          
##                               (1.651)          
##                                                
## cyl                            0.417           
##                               (1.050)          
##                                                
## vs                            10.903*          
##                               (6.058)          
##                                                
## cyl:vs                        -1.724           
##                               (1.051)          
##                                                
## Constant                     22.474***         
##                               (6.060)          
##                                                
## -----------------------------------------------
## Observations                    32             
## R2                             0.830           
## Adjusted R2                    0.789           
## Residual Std. Error       2.766 (df = 25)      
## F Statistic           20.357*** (df = 6; 25)   
## ================================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

You probably have noticed that `+` in a formula does not mean literal additiona, and `*` does not mean literal multiplication. Both operators have been overloaded in the context of formulas. The effect of `*` here is to include terms in the model for both the variables `cyl` and `vs` separately and a term representing the product of these two variables. If we wanted *literal* multiplication in the model, we would have to encapsulate that multiplication with the `I()` function. The model below is equivalent to `fit3`.

```
stargazer(lm(mpg ~ disp + hp + am + cyl + vs + I(cyl * vs), data = mtcars),
          type = "text")
```

```
##
## ================================================
##                          Dependent variable:
##                      ---------------------------
##                                  mpg
## ------------------------------------------------
## disp                           -0.010
##                                (0.011)
##
## hp                            -0.038**
##                                (0.015)
##
## am                             3.867**
##                                (1.651)
##
## cyl                             0.417
##                                (1.050)
##
## vs                             10.903*
##                                (6.058)
##
## I(cyl * vs)                    -1.724
##                                (1.051)
##
## Constant                      22.474***
##                                (6.060)
##
## ------------------------------------------------
## Observations                      32
## R2                              0.830
## Adjusted R2                     0.789
## Residual Std. Error        2.766 (df = 25)
## F Statistic             20.357*** (df = 6; 25)
## ===============================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

Perhaps we should incorporate cylinders in a different way, though; perhaps cylinders should be treated as categorical variables. This could allow for non-linear effects in the number of cylinders. We could force cylinders to be treated as categorical variables by passing them as factors to the model, like so (note that if the variable `cyl` was already a factor variable, `as.factor()` would not be needed):

```r
fit4 <- lm(mpg ~ disp + hp + am + as.factor(cyl), data = mtcars)
stargazer(fit4, type = "text")
```

```
##
## ===============================================
##                         Dependent variable:
##                     ---------------------------
##                                 mpg
## -----------------------------------------------
## disp                          -0.015
##                               (0.011)
##
## hp                           -0.039**
##                               (0.015)
##
## am                            3.334**
##                               (1.364)
##
## as.factor(cyl)6               -3.222*
##                               (1.589)
##
## as.factor(cyl)8               -1.011
##                               (3.033)
##
## Constant                     29.004***
##                               (1.845)
##
## -----------------------------------------------
## Observations                    32
## R2                             0.837
## Adjusted R2                    0.806
## Residual Std. Error       2.653 (df = 26)
## F Statistic           26.790*** (df = 5; 26)
## ===============================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

Linear models handle categorical variables by producing a dummy variable for the possible values of the categorical variable. Specifically, all categories but one get a dummy variable that indicates whether the observation was in that category or not. One category is omitted because otherwise the dummy variables would become collinear with the intercept term; the group that does not get a dummy variable become the **baseline** group. This is because the coefficients of the other variables are effectively contrasts with the mean of the baseline group. This follows from the interpretation of the coefficients of dummy variables: they are the change in expected value when that variable is true or not. If we did not want a baseline group, we would need to suppress the constant; then all groups in the categorical variable will get a dummy variable.

```
stargazer(lm(mpg ~ disp + hp + am + as.factor(cyl) - 1, data = mtcars),
          type = "text")
```

```
##
## ===============================================
##                         Dependent variable:
##                     ---------------------------
##                                 mpg
## -----------------------------------------------
## disp                          -0.015
##                               (0.011)
##
## hp                           -0.039**
##                               (0.015)
##
## am                            3.334**
##                               (1.364)
##
## as.factor(cyl)4              29.004***
##                               (1.845)
##
## as.factor(cyl)6              25.782***
##                               (2.542)
##
## as.factor(cyl)8              27.993***
##                               (4.237)
##
## -----------------------------------------------
## Observations                    32
## R2                             0.987
## Adjusted R2                    0.984
## Residual Std. Error       2.653 (df = 26)
## F Statistic           328.109*** (df = 6; 26)
## ===============================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

The coefficients of the factor variables in the new model can be interpreted as the mean MPG for each count of cylinders, without yet accounting for the effects of the other variables in the model.

Inference and Model Selection

All the statistical procedures discussed with simple linear regression apply in the multivariate setting, including the diagnostic tools, but some issues should be updated.

First, regarding the F-test. The F-test as described before still works as marketed, but we can make additional F tests to help decide between models. For instance, we could decide between

$$H_0 : \beta_{k^*} = \beta_{k^*+1} = \cdots = \beta_k = 0$$

and

$$H_A : H_0 \text{ is false}$$

for some $k^* \geq 1$. In other words, we can test whether a collection of new regressors have predictive power when added to a smaller linear model.

We may, for example, want to test whether any of the coefficients in `fit4` are non-zero, comparing against the model contained in `fit`. To perform this test, we should use the `anova()` function (another function that can perform `anova()`) like so:

```
anova(fit, fit4)
```

```
## Analysis of Variance Table
##
## Model 1: mpg ~ disp + hp
## Model 2: mpg ~ disp + hp + am + as.factor(cyl)
##   Res.Df    RSS Df Sum of Sq      F   Pr(>F)
## 1     29 283.49
## 2     26 183.04  3    100.45 4.7564 0.008961 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The results of the above test suggest that some of the coefficients in the model `fit4` are not zero; one should necessarily conclude that `fit4` is a better model than `fit`, since `fit` is missing important variables.

Next, let's consider the issue of confidence intervals. `vcov()` still works.

```
vcov(fit4)
```

```
##                  (Intercept)          disp            hp           am
## (Intercept)       3.405382437 -0.0127499444 -0.0067209058 -1.195577901
## disp             -0.012749944  0.0001120357 -0.0000423764  0.006150034
## hp               -0.006720906 -0.0000423764  0.0002208071 -0.009721922
## am               -1.195577901  0.0061500337 -0.0097219220  1.859508755
## as.factor(cyl)6   0.266125814 -0.0052414970 -0.0083459097  0.460108720
## as.factor(cyl)8   2.673529289 -0.0188226829 -0.0231231395  0.792321864
##                 as.factor(cyl)6 as.factor(cyl)8
## (Intercept)          0.266125814      2.67352929
## disp                -0.005241497     -0.01882268
## hp                  -0.008345910     -0.02312314
## am                   0.460108720      0.79232186
## as.factor(cyl)6      2.523821471      3.26500050
## as.factor(cyl)8      3.265000500      9.20011728
```

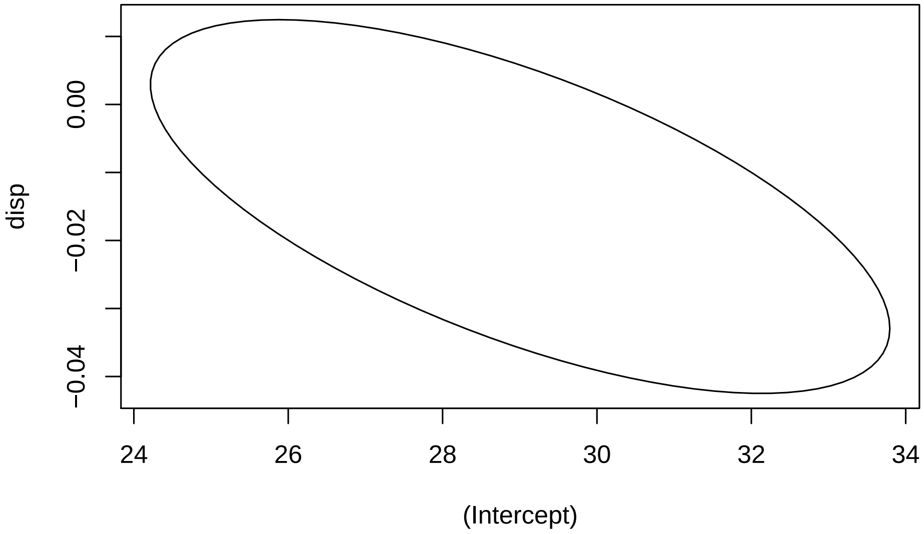
```r
cov2cor(vcov(fit4))
```

```
##                 (Intercept)       disp         hp         am
## (Intercept)      1.00000000 -0.6527504 -0.2450972 -0.4751118
## disp            -0.65275043  1.0000000 -0.2694259  0.4260889
## hp              -0.24509720 -0.2694259  1.0000000 -0.4797848
## am              -0.47511180  0.4260889 -0.4797848  1.0000000
## as.factor(cyl)6  0.09077677 -0.3117079 -0.3535394  0.2123890
## as.factor(cyl)8  0.47764509 -0.5862821 -0.5130310  0.1915604
##                 as.factor(cyl)6 as.factor(cyl)8
## (Intercept)           0.09077677       0.4776451
## disp                 -0.31170793      -0.5862821
## hp                   -0.35353943      -0.5130310
## am                    0.21238901       0.1915604
## as.factor(cyl)6       1.00000000       0.6775748
## as.factor(cyl)8       0.67757481       1.0000000
```

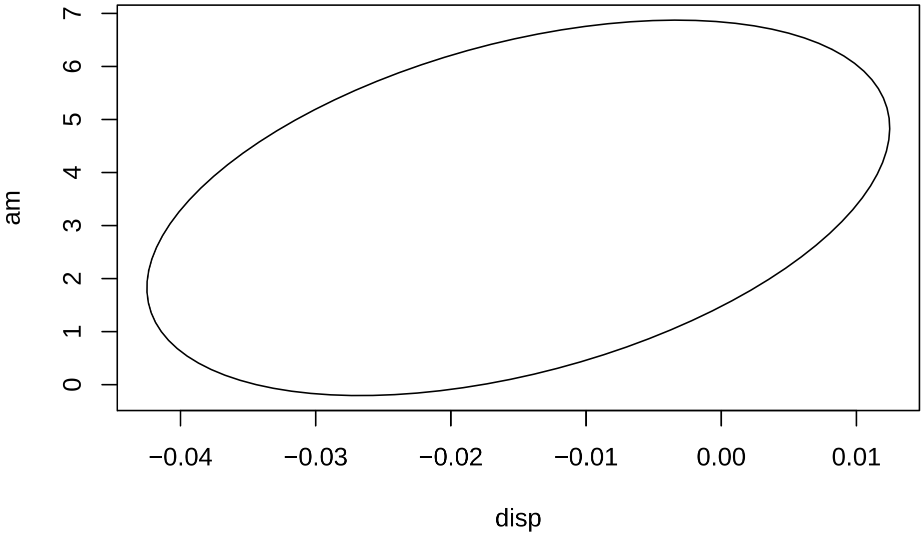
Again, the confidence region is an elliptical space, but this time in $k + 1$ dimensional space, rather than two-dimensional space. Attempting to visualize high-dimensional space directly is known to induce madness, so we may be better off looking at pairs of relationships between coefficients. We can do so by telling the `ellipse()` function we saw earlier which variables we wish to visualize.

```r
library(ellipse)
plot_conf_ellipse <- function(..., col = 1, add = FALSE) {
  if (add) {
    pfunc = lines
  } else {
    pfunc = plot
  }
  pfunc(ellipse(...), type = "l", col = col)
}
```

```r
plot_conf_ellipse(fit4, which = c("(Intercept)", "disp"))
```



```r
plot_conf_ellipse(fit4, which = c("disp", "am"))
```



Suppose we wish to predict values from our model, maybe also get point-wise confidence intervals or prediction intervals. We will need to use the `predict()` function as we did before. For what it's worth, `predict()` works exactly the same; it needs to be fed a `data.frame` containing the new values of the model's regressors, in a format imitating the original data. (You don't need to manually create automatically generated dummy variables, for example, or insert columns for interaction variables.) Visualizing the predictions, though, will be more difficult due to the multivariate setting. (In general, if you want to learn more about multivariate visualization techniques, consider taking a class dedicated to visualization.)

While `predict()` has not changed, my function `predict_func()` should be adapted to the new setting.

```
predict_func_multivar <- function(fit) {
  function(..., interval = "none", level = 0.95) {
    dat <- data.frame(...)
    predict(fit, dat, interval = interval, level = level)
  }
}

mtcar_pf <- predict_func_multivar(fit2)
mtcar_pf(disp = c(160, 150), hp = c(110, 80), am = c(0, 1))
```

```
##        1        2
## 21.06192 26.24082
```

```
mtcar_pf(disp = c(160, 150), hp = c(110, 80), am = c(0, 1),
         interval = "confidence")
```

```
##        fit      lwr      upr
## 1 21.06192 19.06074 23.06309
## 2 26.24082 24.09598 28.38566
```

Now let's consider criteria for deciding between models. This is new to the multivariate context; in simple linear regression there's no interesting model selection discussion to be had. But for multivariate models, we have to decide which coefficients to include or exclude and in what functional form they should appear. At first one may think that one should check the statistical significance of model coefficients and F-tests and use those results to decide what model is most appropriate for the data. This approach is mistaken. First, it's path-dependent; if we choose a different sequence of models to check we may end up with different end results for coefficients. It also doesn't inform us well regarding how we should handle changing test results as we insert and delete coefficients; coefficients that once weren't significant could become significant after inserting or deleting another regressor, or vice versa. Secondly, this is multiple hypothesis testing; the hypothesis tests start to lose their statistical guarantees as we test repeatedly.

Okay, how about checking the value of R^2 or adjusted R^2? While we should be sensitive to the value of R^2, we must use it with great caution. Perfectly valid and acceptable statistical models have low R^2 models, and bad models can have high R^2 values. We have two competing concerns: **underfitting** and **overfitting**. Underfitting means our model has little predictive power and could be improved. Models that have been overfitted, though, have excellent predictive power *in the observed sample*, but say little about the general population and lose their predictive power when we feed them new, out-of-sample data. Attempting to maximize R^2 can easily lead to overfitting, and thus should be avoided. However, a low R^2 *might* indicate underfitting.

We need more tools for model selection. One useful tool is the **Akaike information criterion (AIC)**, a statistic intended to estimate the out-of-sample predictive power of a model. We can obtain the AIC for a particular model using the `AIC()` function.

```
AIC(fit)
```

```
## [1] 168.6186
```

There is no universal interpretation of the AIC itself (at least not one that's not heavily theoretical). The AIC should not be considered in isolation, though; instead, we compare the AIC statistics of competing candidate models, selecting the model that *minimizes* the AIC. We can even get a reasonable interpretation of AICs when comparing two. If we have AIC_1 and AIC_2 for two different models, then we can interpret the quantity

$$\exp\left((AIC_1 - AIC_2)/2\right)$$

as how many more times the model with AIC_2 is likely to be true than the model with AIC_1. The function below performs this kind of analysis:

```
aic_compare <- function(fit1, fit2) {
  exp((AIC(fit1) - AIC(fit2)) / 2)
}

aic_compare(fit, fit4)
```

```
## [1] 54.58818
```

That said, we are generally interested in just finding the model that minimizes the AIC among a class of similar models. The function below can find such a model, when given a list of models.

```
min_aic_model <- function(...) {
  models <- list(...)
  aic_list <- sapply(models, AIC)
  best_model <- models[[which.min(aic_list)]]
  best_model
}

best_fit <- min_aic_model(fit, fit2, fit3, fit4)
stargazer(best_fit, type = "text")
```

```
##
## ===============================================
##                           Dependent variable:
##                       ---------------------------
##                                   mpg
## -----------------------------------------------
## disp                            -0.015
##                                 (0.011)
##
## hp                             -0.039**
##                                 (0.015)
##
## am                              3.334**
##                                 (1.364)
##
## as.factor(cyl)6                -3.222*
##                                 (1.589)
##
## as.factor(cyl)8                -1.011
##                                 (3.033)
##
## Constant                       29.004***
##                                 (1.845)
##
## -----------------------------------------------
## Observations                      32
## R2                               0.837
## Adjusted R2                      0.806
## Residual Std. Error        2.653 (df = 26)
## F Statistic            26.790*** (df = 5; 26)
## ===============================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

```
sapply(list(fit, fit2, fit3, fit4), function(l) {aic_compare(best_fit, l)})
```

```
## [1] 0.01831898 0.25140981 0.18120055 1.00000000
```

It seems that the fourth model seems to best describe the data. That said, some of the coefficients in the model are not statistically different from zero. That's okay; sometimes a good model needs these regressors. However, be aware that model selection via AIC is an asymptotic method; the sample size needs to be large for the AIC to work well.

You may be tempted to write a script that tries just about every combination of parameters and functional forms in a linear model, finds the one with the smallest AIC, and returns it; it's an automatic statistician that would put you out of a job if your employers were aware of it. Resist the temptation! The AIC is a tool to help pick models, but is not a substitute for human domain knowledge. Left to its own devices, the AIC could select models that don't have the best predictive power because they make no sense, and any subject matter expert would say so. The most important model selection tool is human knowledge of the problem. Lean heavily on it! Consider only models that make sense. Consult experts in the subject before building models.

Polynomial Regression

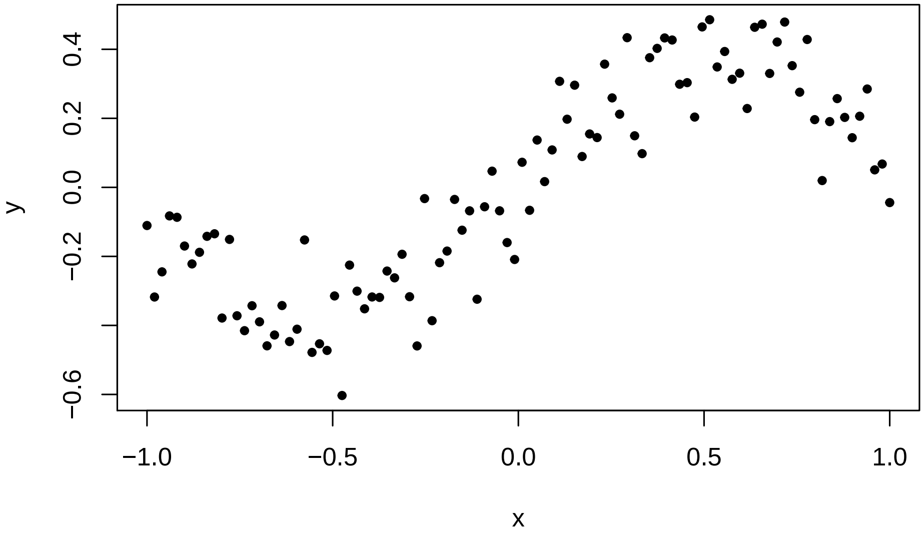
Let's back up and return to the univarate context, where we have one explanatory variable and one response variable. Why should we consider models only of the form $y = a + bx$? Why not quadratic models, $y = a + bx + cx^2$? Or higher-order polynomials, $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_k x^k$? Well, we can in fact consider such models, and we call the topic **polynomial regression**. These are regression models of the form:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_k x_i^k + \epsilon_i.$$

We can view these models as multivariate models; we just add the variables x, x^2, x^3, and so on to our model. These variabes are *not* linearly related to each other, so we can generally estimate such models.

Let's demonstrate by simulating a data set that takes a cubic form on the interval $[-1, 1]$. This will be a cubic function plus some noise.

```r
x <- seq(-1, 1, length = 100)
y <- x - x^3 + rnorm(100, sd = 0.1)
plot(x, y, pch = 20)
```

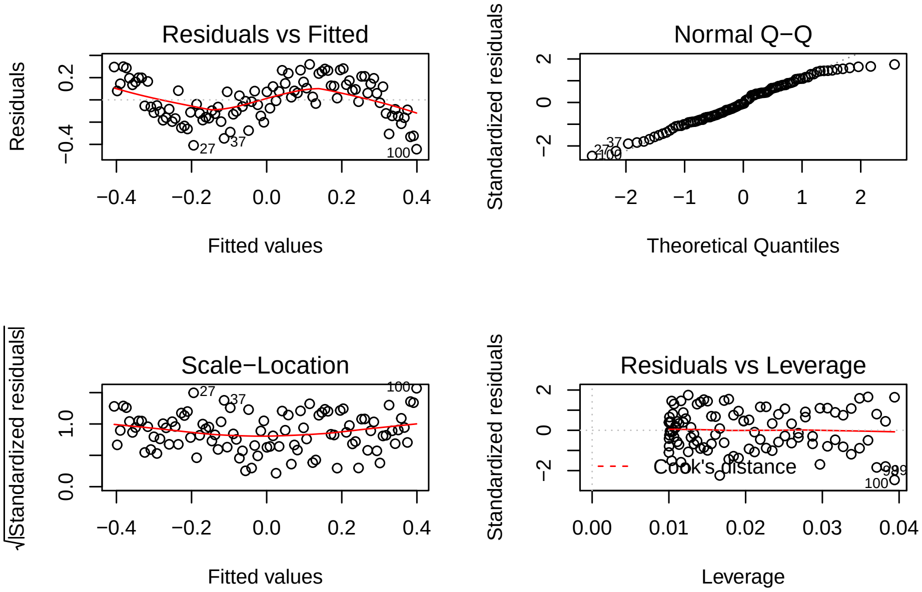


Clearly a model that is just a line is not appropriate for this data set; diagnostic plots would reveal this.

```r
fit <- lm(y ~ x)
summary(fit)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.44324 -0.13298 -0.01006  0.14424  0.31918
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.003444   0.018376  -0.187    0.852
## x            0.402583   0.031511  12.776   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1838 on 98 degrees of freedom
## Multiple R-squared:  0.6248, Adjusted R-squared:  0.621
## F-statistic: 163.2 on 1 and 98 DF,  p-value: < 2.2e-16
```

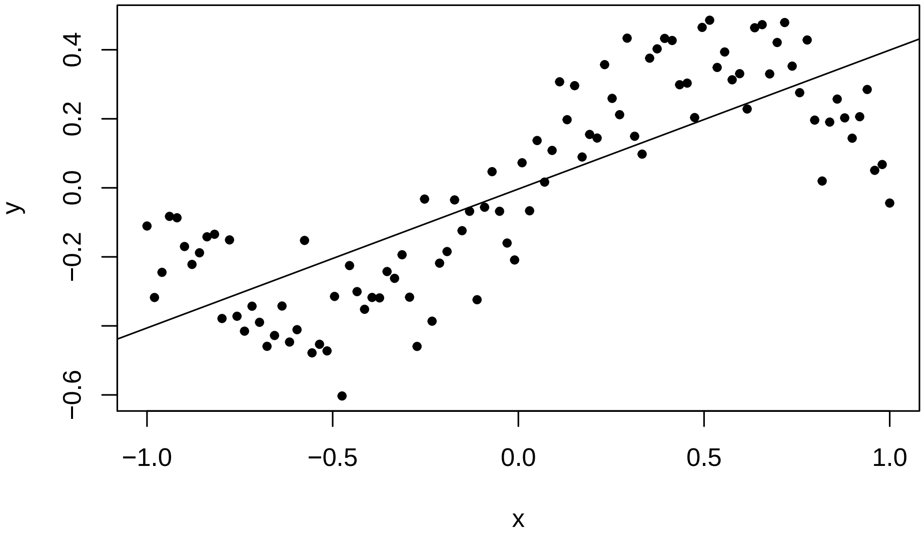
```r
old_par <- par()
par(mfrow = c(2, 2))
plot(fit)
```



```
par(old_par)
```

```
## Warning in par(old_par): graphical parameter "cin" cannot be set

## Warning in par(old_par): graphical parameter "cra" cannot be set

## Warning in par(old_par): graphical parameter "csi" cannot be set

## Warning in par(old_par): graphical parameter "cxy" cannot be set

## Warning in par(old_par): graphical parameter "din" cannot be set

## Warning in par(old_par): graphical parameter "page" cannot be set
```

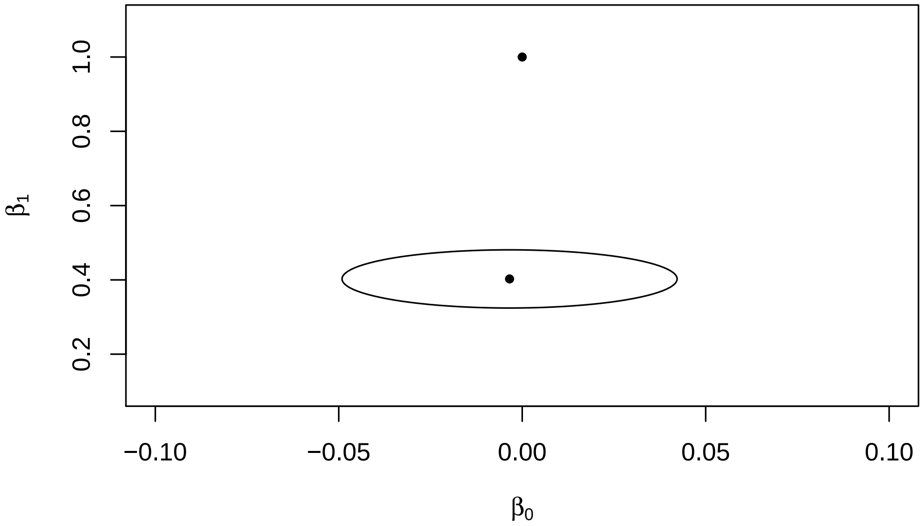
```
plot(x, y, pch = 20)
abline(fit)
```



The problem is that the data should be about evenly distributed around the line, but instead we see data being more likely to be above the line at the start of the window, below in the first half, above in the second half, and below at the very end. This indicates an inappropriate functional form, and we should rectify the issue.

Notice the estimated values? We see that the estimated coefficient for the intercept term is close to zero, and for the linear term close to 0.4. We know that the value of these coefficients in the true model are zero and one, respectively, but these parameters are not even close to the true values.

```
plot(c(0, fit$coefficients[[1]]), c(1, fit$coefficients[[2]]), pch = 20,
     xlim = c(-0.1, 0.1), ylim = c(0.1, 1.1), xlab = expression(beta[0]),
     ylab = expression(beta[1]))
plot_conf_ellipse(fit, add = TRUE)
```



When the model is misspecified this way, the eventual "best" values for β_0 and β_1 are 0 and 0.4 rather than the true values of 0 and 1. That is, if $f(x) = x - x^3$ and $\hat{f}(x; \beta_0, \beta_1) = \beta_0 + \beta_1 x$, the regression model will estimate the values β_0 and β_1 that minimize

$$S(\beta_0, \beta_1) = \int_{-1}^{1} (f(x) - \hat{f}(x; \beta_0, \beta_1))^2 dx = \int_{-1}^{1} (x - x^3 - \beta_0 - \beta_1 x)^2 dx.$$

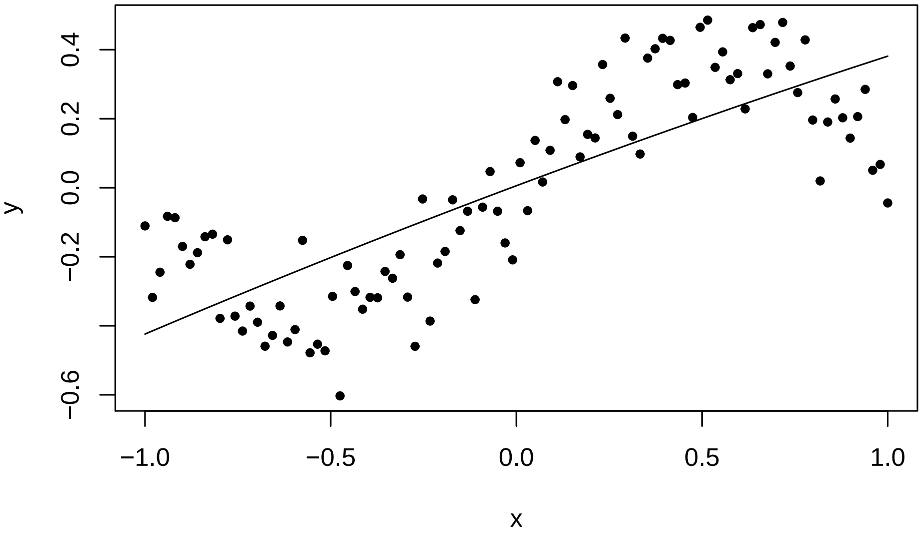
Perhaps if we were to include a quadratic term? We can do so using the `I()` function (we cannot call `^` directly in a formula; this has special meaning in formulas other than exponentiation).

```
fit2 <- lm(y ~ x + I(x^2))

predict_func <- function(fit) {
  response <- names(fit$model)[[1]]
  explanatory <- names(fit$model)[[2]]
  function(x, ...) {
    dat <- data.frame(x)
    names(dat) <- explanatory
    predict(fit, dat, ...)
  }
}

summary(fit2)
```

```
##
## Call:
```

```
## lm(formula = y ~ x + I(x^2))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.4253 -0.1343 -0.0163  0.1473  0.3136
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.005798   0.027679   0.209    0.835
## x            0.402583   0.031640  12.724   <2e-16 ***
## I(x^2)      -0.027175   0.060670  -0.448    0.655
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1845 on 97 degrees of freedom
## Multiple R-squared:  0.6256, Adjusted R-squared:  0.6179
## F-statistic: 81.05 on 2 and 97 DF,  p-value: < 2.2e-16
```

```r
fit2_func <- predict_func(fit2)
plot(x, y, pch = 20)
lines(x, fit2_func(x))
```

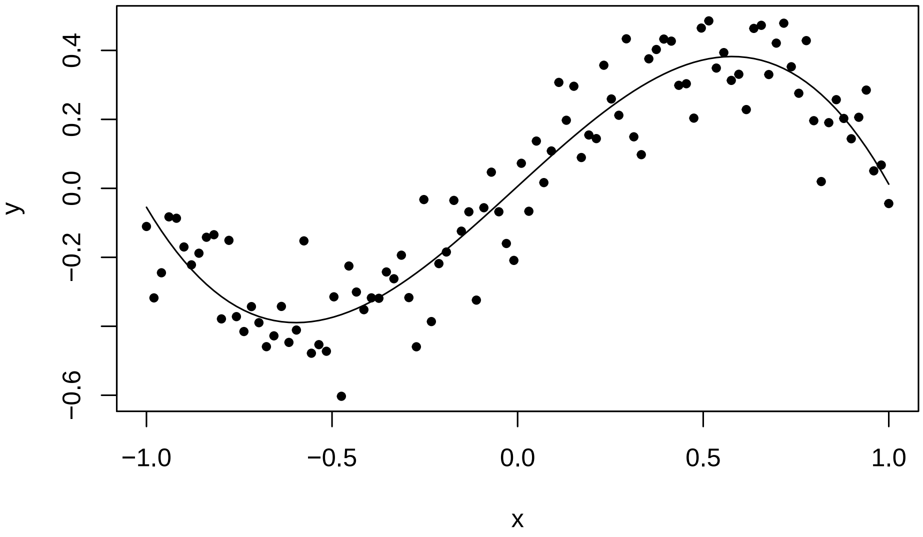


Unfortunately the quadratic model is not a big improvement. But did you notice that the resulting predicted line has a slight bend? There was an effort to fit the line but it didn't quite work. But of course it didn't; the model is incorrect, and the true value of the quadratic coefficient is zero. What we should do is insert a cubic term.

```r
fit3 <- lm(y ~ x + I(x^2) + I(x^3))
summary(fit3)
```

```
##
## Call:
## lm(formula = y ~ x + I(x^2) + I(x^3))
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.252654 -0.058793  0.004324  0.074405  0.236314
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.005798   0.016115   0.360    0.720
## x            0.984863   0.046070  21.377   <2e-16 ***
## I(x^2)      -0.027175   0.035324  -0.769    0.444
## I(x^3)      -0.951376   0.068994 -13.789   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1074 on 96 degrees of freedom
## Multiple R-squared:  0.8744, Adjusted R-squared:  0.8705
## F-statistic: 222.8 on 3 and 96 DF,  p-value: < 2.2e-16
```

```
fit3_func <- predict_func(fit3)
plot(x, y, pch = 20)
lines(x, fit3_func(x))
```



Now our model has a good fit to the data, and the estimated coefficients are close to the truth.

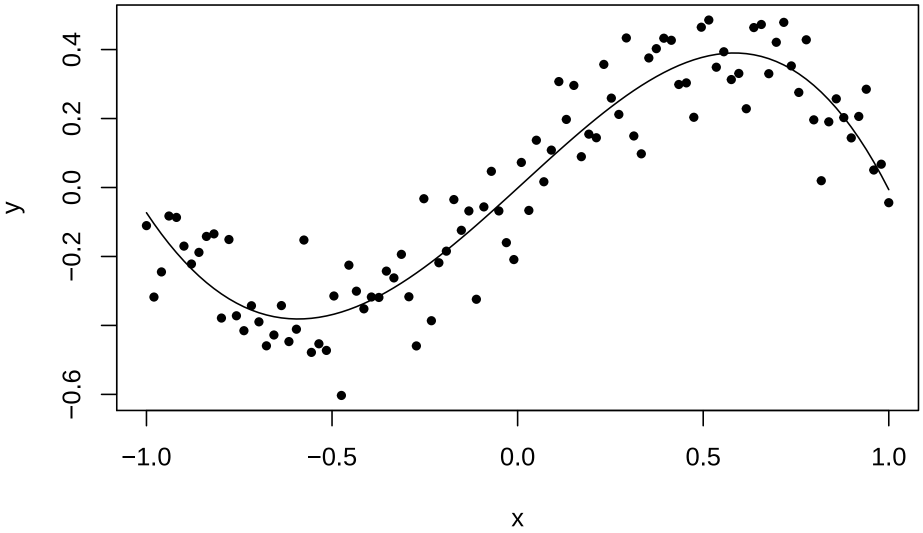
Should we have included the quadratic term even when we believed it might be wrong? The answer is yes. In general we won't know what the true model is so we must include all polynomial orders up to the highest order k.

Okay, but by that same reasoning we don't know that there isn't a fourth-order term, or even higher, in our polynomial. Should we start including those terms too? Let's see what happens when we do.

```r
fit4 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4))
summary(fit4)
```

```
##
## Call:
## lm(formula = y ~ x + I(x^2) + I(x^3) + I(x^4))
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.256735 -0.060422  0.001926  0.079165  0.228586
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.001794   0.020211  -0.089    0.929
## x            0.984863   0.046217  21.310   <2e-16 ***
## I(x^2)       0.047272   0.124120   0.381    0.704
## I(x^3)      -0.951376   0.069214 -13.746   <2e-16 ***
## I(x^4)      -0.085164   0.136077  -0.626    0.533
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1078 on 95 degrees of freedom
## Multiple R-squared:  0.8749, Adjusted R-squared:  0.8696
## F-statistic: 166.1 on 4 and 95 DF,  p-value: < 2.2e-16
```

```r
fit4_func <- predict_func(fit4)
plot(x, y, pch = 20)
lines(x, fit4_func(x))
```

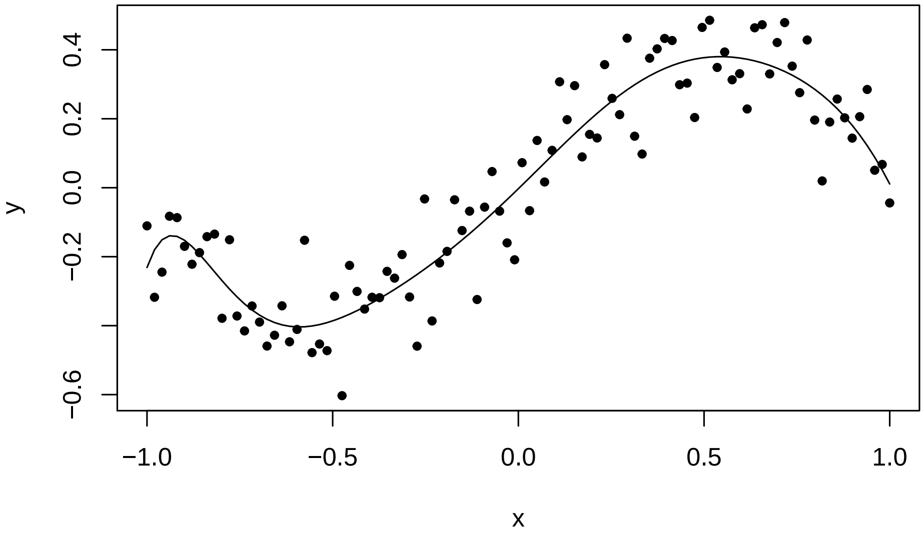


```
fit10 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7) +
                I(x^8) + I(x^9) + I(x^10))
summary(fit10)
```

```
##
## Call:
## lm(formula = y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) +
##     I(x^7) + I(x^8) + I(x^9) + I(x^10))
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.249021 -0.060430  0.002379  0.070313  0.250926
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.002911   0.029244  -0.100    0.921
## x            1.047155   0.167413   6.255 1.35e-08 ***
## I(x^2)       0.278990   0.836658   0.333    0.740
## I(x^3)      -1.362420   1.868061  -0.729    0.468
## I(x^4)      -1.989767   6.135962  -0.324    0.746
## I(x^5)       1.511401   6.592462   0.229    0.819
## I(x^6)       3.984659  17.075153   0.233    0.816
## I(x^7)      -2.856686   8.983226  -0.318    0.751
## I(x^8)      -2.335075  19.928445  -0.117    0.907
## I(x^9)       1.781664   4.139136   0.430    0.668
## I(x^10)     -0.046110   8.226474  -0.006    0.996
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

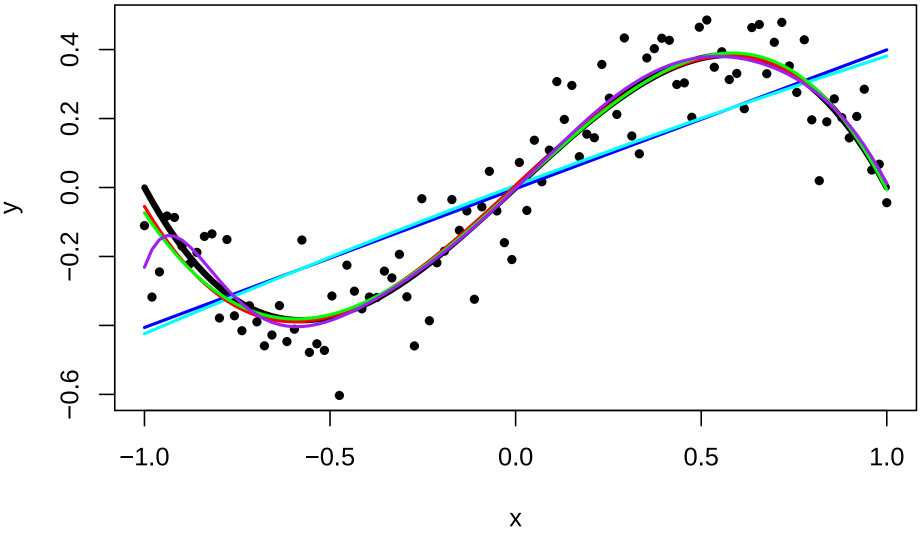
```
## Residual standard error: 0.1079 on 89 degrees of freedom
## Multiple R-squared:  0.8825, Adjusted R-squared:  0.8693
## F-statistic: 66.85 on 10 and 89 DF,  p-value: < 2.2e-16
```

```
fit10_func <- predict_func(fit10)
plot(x, y, pch = 20)
lines(x, fit10_func(x))
```



The quality of the fit is degrading. In my opinion, the last function is clearly not the same as the true function. Below is a plot containing all three functions, with the true function shown as a thick black line.

```
plot(x, y, pch = 20)
lines(x, x - x^3, lwd = 4)
fit1_func <- predict_func(fit)
lines(x, fit1_func(x), col = "blue", lwd = 2)
lines(x, fit2_func(x), col = "cyan", lwd = 2)
lines(x, fit3_func(x), col = "red", lwd = 2)
lines(x, fit4_func(x), col = "green", lwd = 2)
lines(x, fit10_func(x), col = "purple", lwd = 2)
```



Eventually, if we were to keep adding higher-order terms to the polynomial, we would have an *exact* fit to the data, where $\hat{y}_i = y_i$. This is *not* good since the resulting model would be a monstrosity far from the truth and with no predictive power at all. Thus, when fitting polynomials to data, we want exactly as many polynomial terms as we need and not one more.

That said, what if the true function is not a polynomial? I have exciting news: *polynomial regression can fit non-polynomial functions on compact (finite) intervals!* The reasons are beyond the scope of this course (go learn more analysis and linear algebra), but any continuous, smooth function can be approximated well by polynomials on compact intervals. Your job, as statistician, is to decide how many polynomial terms you need. Too few and you have a bad fit for the true function; too many and you risk overfitting. But tools such as the AIC can be used to select the proper order of polynomial regression, too.

Here's what the AIC has to say about the models considered here:

```r
sapply(list(fit, fit2, fit3, fit4, fit10), AIC)
```

```
## [1]  -51.06253  -49.26915 -156.48390 -154.89536 -149.15537
```

The AIC was minimized for model 3, the third-order polynomial and the correct model order. This of course is not the end of polynomial order selection, but a good first step.

Lecture 11

Two-Way ANOVA

ANOVA was presented as a way to determine if the means of all populations under consideration are the same. One way in which this idea manifests is as testing whether a single set of mutually exclusive treatments have the same effect on subjects or not. Researchers, though, would like to be able to apply multiple classes of treatments to subjects.

For example, let's consider the `ToothGrowth` data set.

```
head(ToothGrowth)
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

On the one hand, different guinea pigs will be given different types of supplements: orange juice (`OJ`) or vitamin C (`VC`). On the other hand, different dosages are administered to the guinea pigs, either half, full, or double dose. We could view these as two different kinds of treatments, which are not mutually exclusive from each other. In terms of populations, we might say that while there are different types of populations, we have classes of population types and there can be overlap of populations when population types come from different classes.

We first need to ask: are there interactions among treatments? Non-interaction effectively means that the effects of different treatments in some sense "add up". For example, doubling the dose of vitamin C has relatively the same effect as doubling the dose of orange juice. But interaction would occur if, say, doubling the dose of vitamin C would cause a reduction in tooth growth while doubling the dose of orange juice caused an increase. A model including interactions is harder to interpret than a model without one, though both can be handled.

If we don't have interactions in the model, then the two-way ANOVA model looks like so:

$$x_{ijk} = \mu_j + \nu_k + \epsilon_{ijk}.$$

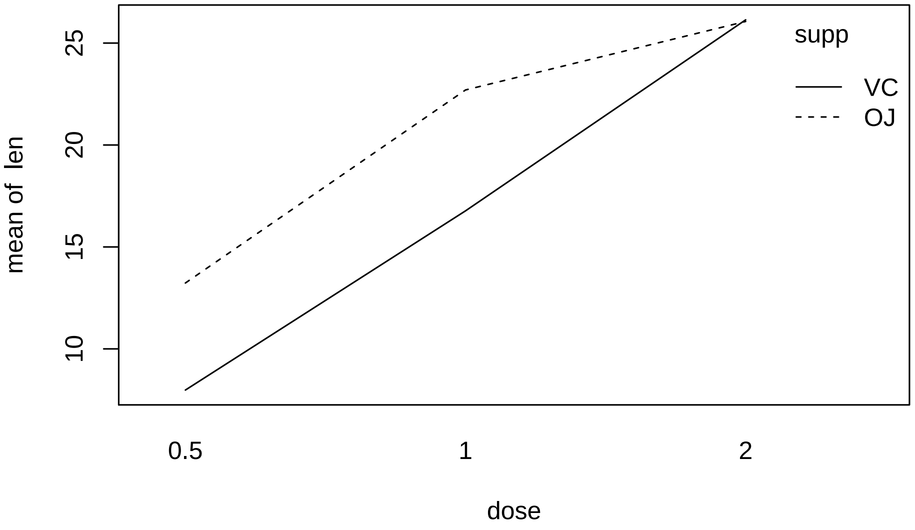
In this model μ_j is the effect of treatment j out of J total treatments in that class while ν_k is the effect of treatment k out of K total treatments in that class. A two-way ANOVA model with interactions would look like so:

$$x_{ijk} = \mu_j + \nu_k + \gamma_{jk} + \epsilon_{ijk}.$$

The γ_{jk} term is the interaction effect for that combination of populations. Aside from these issues; two-way ANOVA makes the same assumptions as one-way ANOVA: Normally distributed residuals with a constant variance.

We can try to determine if interaction effects exist by using an interaction plot. Here is an interaction plot for the `ToothGrowth` data set.

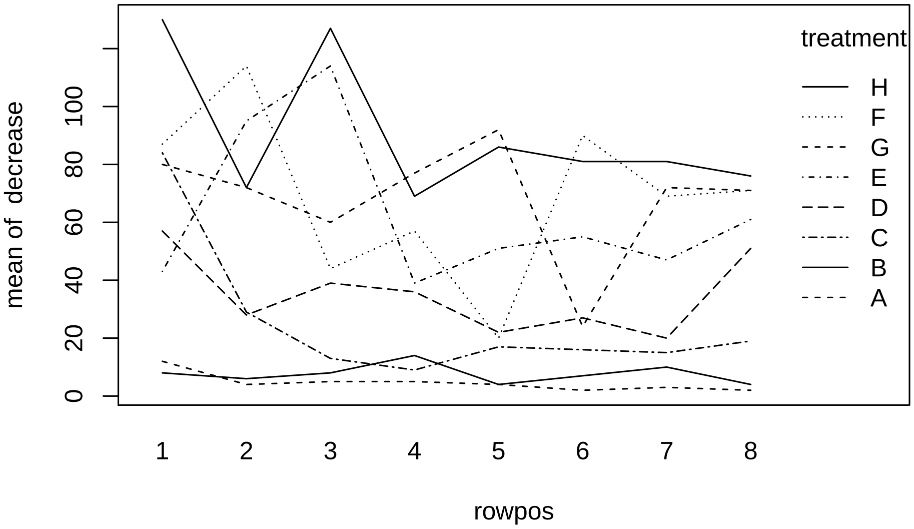
```
with(ToothGrowth, interaction.plot(dose, supp, len))
```



The x-axis tracks dosage while the y-axis tracks the mean of the variable `len`, the amount of tooth growth in a guinea pig. Different lines are drawn for `supp`, or the type of supplement used. We are looking to see if these lines cross. A crossing of lines suggests that an increase of dosage for one type of supplement has a different effect than it does for the other dosage. Ideally we would also have roughly parallel lines. Here there may be a tiny crossing at double dosage, though not dramatic. The lines are not quite parallel, unfortunately. It seems that this interaction plot could be interpreted either way. If one were being conservative, they would likely treat the data as having interactions, since a model allowing for interactions can cope if there are no interactions in truth; the opposite direction is harder to say.

Let's compare with the `OrchardSprays` data set, which contains the result of an experiment assessing the potency of different orchard sprays in repelling honeybees. If one treatment was the type of spray and the other the row position of where the spray was used, we would get the following interaction plot:

```r
with(OrchardSprays, interaction.plot(rowpos, treatment, decrease))
```



There absolutely are interactions among the factors so a two-way ANOVA model without interactions certainly would be inappropriate (though I should mention that a different design was intended for the experiment).

We would like to estimate the two-way ANOVA model, either with or without interactions. However, we can understand ANOVA, in general, as linear regression with the regressors being dummy variables. We would have a sequence of dummy variables for one class of treatments and a sequence of dummy variables for another class of treatments. Thus we can use `lm()` to estimate two-way ANOVA models like so:

```r
(tf1 <- lm(len ~ supp + as.factor(dose), data = ToothGrowth))
```

```
##
## Call:
## lm(formula = len ~ supp + as.factor(dose), data = ToothGrowth)
##
## Coefficients:
##      (Intercept)           suppVC  as.factor(dose)1  as.factor(dose)2
##            12.45            -3.70              9.13             15.49
```

(Notice we have to tell `lm()` we want `dose` to be treated as a factor variable;

otherwise, it would be treated as quantitative, which we don't want.) The resulting model is phrased in terms of contrasts; the baseline group consists of guinea pigs that got half a dose of vitamin C. Below we do some statistics for the model.

```r
summary(tf1)
```

```
## 
## Call:
## lm(formula = len ~ supp + as.factor(dose), data = ToothGrowth)
## 
## Residuals:
##    Min     1Q Median     3Q    Max 
## -7.085 -2.751 -0.800  2.446  9.650 
## 
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)    
## (Intercept)        12.4550     0.9883  12.603  < 2e-16 ***
## suppVC             -3.7000     0.9883  -3.744 0.000429 ***
## as.factor(dose)1    9.1300     1.2104   7.543 4.38e-10 ***
## as.factor(dose)2   15.4950     1.2104  12.802  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 3.828 on 56 degrees of freedom
## Multiple R-squared:  0.7623, Adjusted R-squared:  0.7496 
## F-statistic: 59.88 on 3 and 56 DF,  p-value: < 2.2e-16
```

The F-test performed can be viewed as a statistical test deciding between the null hypothesis of no treatments having any effect on the response variables and the alternative that at least one treatment has an effect on the response variable. In this case the null hypothesis is rejected; there appears to be some treatment effect, either among the supplement type or the dosage. Furthermore, by examining the results of the tests for the coefficients, just about every possible type of treatment has some effect.

How about allowing for interactions? We can introduce interaction terms like so:

```r
(tf2 <- lm(len ~ supp * as.factor(dose), data = ToothGrowth))
```

```
## 
## Call:
## lm(formula = len ~ supp * as.factor(dose), data = ToothGrowth)
## 
## Coefficients:
##             (Intercept)                   suppVC         as.factor(dose)1  
##                   13.23                    -5.25                     9.47  
##        as.factor(dose)2  suppVC:as.factor(dose)1  suppVC:as.factor(dose)2  
##                   12.83                         -0.68                          5.33
```

```r
summary(tf2)
```

```
##
## Call:
## lm(formula = len ~ supp * as.factor(dose), data = ToothGrowth)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##  -8.20  -2.72  -0.27   2.65   8.27
##
## Coefficients:
##                         Estimate Std. Error t value Pr(>|t|)
## (Intercept)               13.230      1.148  11.521 3.60e-16 ***
## suppVC                    -5.250      1.624  -3.233  0.00209 **
## as.factor(dose)1           9.470      1.624   5.831 3.18e-07 ***
## as.factor(dose)2          12.830      1.624   7.900 1.43e-10 ***
## suppVC:as.factor(dose)1   -0.680      2.297  -0.296  0.76831
## suppVC:as.factor(dose)2    5.330      2.297   2.321  0.02411 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.631 on 54 degrees of freedom
## Multiple R-squared:  0.7937, Adjusted R-squared:  0.7746
## F-statistic: 41.56 on 5 and 54 DF,  p-value: < 2.2e-16
```

The results of the new model are more difficult to interpret, both literally and statistically. Additionally, only one potential interaction appears to possibly be statistically significant: the interaction between the vitamin C supplement and the dosage amount.

This begs the question: should we include interaction terms for dosage and supplement or not? So far the results are not particularly conclusive. What we should do is decide whether the coefficients for the interaction terms are statistically different from zero or not. And while we are at it, we should determine whether each type of treatment *individually* has an effect (not just whether any treatment at all has an effect).

We can do this by using the F tests for comparing different linear models, with one model being a particular instance of the other, as we did in the last lecture. First we should introduce models representing one-way ANOVA models tracking just supplement type and dosage, like so:

```r
(tfs <- lm(len ~ supp, data = ToothGrowth))
```

```
##
## Call:
```

```
## lm(formula = len ~ supp, data = ToothGrowth)
##
## Coefficients:
## (Intercept)       suppVC
##       20.66        -3.70
```

```
(tfd <- lm(len ~ as.factor(dose), data = ToothGrowth))
```

```
##
## Call:
## lm(formula = len ~ as.factor(dose), data = ToothGrowth)
##
## Coefficients:
##      (Intercept)  as.factor(dose)1  as.factor(dose)2
##            10.61              9.13             15.49
```

Then we compare the model that includes terms for both supplement and dosage to one of these two models. If adding additional terms for another treatment have non-zero coefficients, then the other treatment has an effect on the response variable.

```
anova(tfs, tf1)  # Reject H0: Dosage has an effect
```

```
## Analysis of Variance Table
##
## Model 1: len ~ supp
## Model 2: len ~ supp + as.factor(dose)
##   Res.Df    RSS Df Sum of Sq      F    Pr(>F)
## 1     58 3246.9
## 2     56  820.4  2    2426.4 82.811 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
anova(tfd, tf1)  # Reject H0: Supplement type has an effect
```

```
## Analysis of Variance Table
##
## Model 1: len ~ as.factor(dose)
## Model 2: len ~ supp + as.factor(dose)
##   Res.Df     RSS Df Sum of Sq      F    Pr(>F)
## 1     57 1025.78
## 2     56  820.43  1    205.35 14.017 0.0004293 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

If we want to see if there are interactions among dosage type and supplement, we can do so by comparing the model with interactions to the one without.

```
anova(tf1, tf2)  # Reject H0: Interactions
```

```
## Analysis of Variance Table
##
## Model 1: len ~ supp + as.factor(dose)
## Model 2: len ~ supp * as.factor(dose)
##   Res.Df    RSS Df Sum of Sq     F  Pr(>F)
## 1     56 820.43
## 2     54 712.11  2    108.32 4.107 0.02186 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

It does seem that there are interactions between dosage and supplement types.

ANOVA models are a particular class of linear model and can be understood as such. ANOVA differs from linear regression mostly in presentation. That said, people care a great deal about presentation, so let's see how we can perform two-way ANOVA with some of the other functions we've seen.

The `aov()` function we saw earlier is well-equipped for two-way ANOVA, and can be called basically just like `lm()`. The advantage to using `aov()` is that it's specifically designed for ANOVA.

```
aov(len ~ supp + as.factor(dose), data = ToothGrowth)  # No interactions
```

```
## Call:
##    aov(formula = len ~ supp + as.factor(dose), data = ToothGrowth)
##
## Terms:
##                     supp as.factor(dose) Residuals
## Sum of Squares   205.350        2426.434   820.425
## Deg. of Freedom        1               2        56
##
## Residual standard error: 3.82759
## Estimated effects may be unbalanced
```

```
aov(len ~ supp * as.factor(dose), data = ToothGrowth)  # Interactions
```

```
## Call:
##    aov(formula = len ~ supp * as.factor(dose), data = ToothGrowth)
##
## Terms:
##                     supp as.factor(dose) supp:as.factor(dose) Residuals
## Sum of Squares   205.350        2426.434              108.319   712.106
## Deg. of Freedom        1               2                    2        54
##
## Residual standard error: 3.631411
## Estimated effects may be unbalanced
```

Lecture 12

Nonparametric Statistics

Nonparametric statistics concern statistical inference while making few assumptions about the underlying distribution of the data. Note that this does not mean there are *no* assumptions about the data; rather, it means we are generally not assuming that the data comes from specific family of distributions. Common assumptions for nonparametric methods are assumptions such as:

- The data was drawn from a continuous (i.e. not discrete) distribution.
- The distribution from which the data was drawn is symmetric.

Throughout this lecture the first assumption is generally assumed; these methods don't work as well with discrete data as they do with continuous data. The second assumption may be assumed for certain tests.

Nonparametric methods' commonly work with the quantiles of the data. Quantities such as the mean or variance may or may not exist for certain distributions. However, quantiles or generalized notions of quantiles can always be defined for any random variable, and for continuous random variables a unique number can be assigned to every quantile. Since quantiles always exist, we can always perform inference for them while making weak assumptions about the data.

Some will view a parametric test such as the sign test, sign-rank test, or the Wilcoxon rank-sum test as equivalent to their parametric cousins such as the t-test. This is not true since the nonparametric tests generally check quantiles such as the median while t-tests and z-tests are tests for the mean. (Granted, the mean and median of the Normal distribution or any symmetric distribution are the same, but in general they are not necessarily the same.) If the research question at hand is specifically for the mean and the data is not assumed to be symmetric, then these nonparametric tests are inappropriate. The reverse is also true; the z-test should not be used for inference about the median when the data is not symmetric (the t-test is automatically inappropriate due to non-Normality, but it's equivalent to the z-test for large sample sizes).

Sign Test

Let q_p be the $100p^{\text{th}}$ percentile of the data; automatically $q_{0.5}$ is the median of the data. We wish to decide between the hypotheses:

$$H_0 : q_p = q_{p0}$$

$$H_A : \begin{cases} q_p > q_{p0} \\ q_p \neq q_{p0} \\ q_p < q_{p0} \end{cases}$$

Let's continue this discussion but specifically for the median. In order for a number to be the median of a random variable, the probability that the random variable exceeds that number needs to be 0.5. So if q_{p0} with $p = 0.5$ were in fact the median, then $P(X_i > q_{p0}) = p = 0.5$. If the true median was not q_{p0} though this would not be true. If in fact the true median were greater than q_{p0} then $P(X_i > q_{p0}) > p = 0.5$. The converse could also be said; if the true median were less than q_{p0} then $P(X_i > q_{p0}) < p = 0.5$. This suggests that what we should be tracking is whether an observation in the sample exceeds q_{p0} or not. (If an observation exactly equals q_{p0}, delete it.)

If T is a statistic that counts the number of times an observation exceeds q_{p0} then the if the null hypothesis is true the distribution of this statistic is known; it counts the number of times the median is exceeded (a "success") or not (a "failure"), and thus follows a $\text{Bin}(n, p)$ distribution. If the alternative hypothesis states that the true median is greater than q_{p0} then large T would be evidence in favor of the alternative; this would determine how we compute p-values. Similar statements would be made for the other possible alternative hypotheses. Ultimately the test reduces to a test for population proportion, where the original continuous data is converted to binary data tracking whether the median under the null hypothesis was exceeded or not; hence the term "sign test" since we're tracking the sign of $X_i - q_{p0}$.

The function below implements the sign test.

```r
sign.test <- function(x, q = 0, p = 0.5, alternative = "two.sided") {
  res <- list()
  res$data.name <- deparse(substitute(x))
  res$estimate <- c("quantile" = quantile(x, p)[[1]])
  x <- x[x != q]  # Delete observations matching q exactly
  res$method <- "Sign Test"
  res$parameter <- c("p" = p)
  res$alternative <- alternative
  res$null.value <- c("quantile" = q)
  res$statistic <- c("T" = sum(x > q))
  n <- length(x)
  res$p.value <- binom.test(res$statistic, n, p = 1 - p,
                            alternative = alternative)$p.value
  class(res) <- "htest"
  res
}
```

We will demonstrate its use on simulated data.

```
(x <- rcauchy(10, location = 3, scale = 2))  # Cauchy distributed; t-test won't
```

```
## [1]  4.627787  2.288983  1.077072  3.205023  3.193945  3.896991 23.270197
## [8]  3.555012  4.521352  3.671545
```

```
                                             # work
sign.test(x, q = 3)
```

```
##
##  Sign Test
##
## data:  x
## T = 8, p = 0.5, p-value = 0.1094
## alternative hypothesis: true quantile is not equal to 3
## sample estimates:
## quantile
## 3.613279
```

```
sign.test(x, q = 2, alternative = "greater")
```

```
##
##  Sign Test
##
## data:  x
## T = 9, p = 0.5, p-value = 0.01074
## alternative hypothesis: true quantile is greater than 2
## sample estimates:
## quantile
## 3.613279
```

```
sign.test(x, q = 2, alternative = "less")
```

```
##
##  Sign Test
##
## data:  x
## T = 9, p = 0.5, p-value = 0.999
## alternative hypothesis: true quantile is less than 2
## sample estimates:
## quantile
## 3.613279
```

```
sign.test(x, q = 2, p = 0.25, alternative = "less")  # Testing quartile
```

```
##
##  Sign Test
##
## data:  x
## T = 9, p = 0.25, p-value = 0.9437
## alternative hypothesis: true quantile is less than 2
## sample estimates:
## quantile
## 3.196715
```

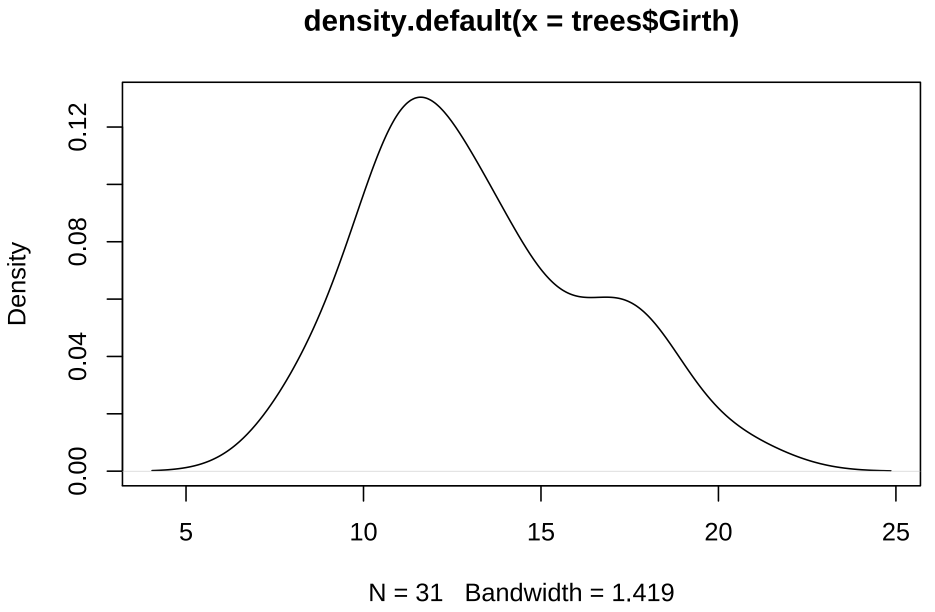
Signed-Rank Test

The Wilcoxon signed-rank test is a non-parametric test for the median specifically (as opposed to a general quantile) and is intended for data coming from a symmetric distribution. Before using it, you should check this assumption; look at histograms or density estimates and decide if the data appears reasonably symmetric. The test is intended to have better power than the sign test while not going as far as the t-test in assuming that the data is Normally distributed. Since we are restricting ourselves to symmetric distributions, the signed-rank test is equivalent to the t-test when the population has finite and well-defined mean and variance; hopefully the test will have better power than the t-test, but this is not guaranteed even for non-Normal data (the t-test is the most powerful test when data is Normally distributed).

We will use the notation $q_{0.5} = m$. We have the same null and alternative hypotheses as before, but the test statistic not only accounts for whether the data is greater than the median or not (the "signed" part) but also the *rank* of the data, where one ranks observations by how far away they are from the supposed median (so the closest observation has a rank of 1 and the furthest a rank of n). The test statstic will be $T = \sum_{i:x_i > m_0} \text{rank}(|x_i - m_0|)$. Suppose that the alternative hypothesis says that the true median is greater than the median under the null hypothesis. Then large T would serve as evidence against the null hypothesis as observations above the median also tend to be some of the most distant. We can come up with rejection regions for other alternative hypotheses, and the distribution of the statistic under the null hypothesis is known (but not necessarily simple). Thus we can do tests.

The R function for this test is `wilcox.test()`, with parameters similar to `t.test()`. Here for example is a demonstration of `wilcox.test()` to determine whether the median of the girth of trees is 12 or not.

```
plot(density(trees$Girth))  # Symmetric enough
```



```
wilcox.test(trees$Girth, mu = 12)
```

```
## Warning in wilcox.test.default(trees$Girth, mu = 12): cannot compute exact
## p-value with ties

## Warning in wilcox.test.default(trees$Girth, mu = 12): cannot compute exact
## p-value with zeroes

##
##  Wilcoxon signed rank test with continuity correction
##
## data:  trees$Girth
## V = 323.5, p-value = 0.06263
## alternative hypothesis: true location is not equal to 12
```

Wilcoxon Rank-Sum Test

The two-sample t-test helps decide whether the means of two populations are the same or not. The nonparametric equivalent of the two-sample t-test is the Wilcoxon rank-sum test. The test works for two distributions that are identical except for the location of the median. Let m_X be the median of one population and m_Y the median of the other. Then our null hypothesis says:

$$H_0 : m_X = m_Y$$

Our alternative is of the form:

$$H_A : \begin{cases} m_X > m_Y \\ m_X \neq m_Y \\ m_X < m_Y \end{cases}$$

The R function for this test is `wilcox.test()` and takes two data sets. The parameter `alternative` can be used to set which alternative hypothesis is tested.

```r
# First, let's get the data into separate vectors
split_len <- split(ToothGrowth$len, ToothGrowth$supp)
OJ <- split_len$OJ
VC <- split_len$VC
# Perform statistical test
wilcox.test(OJ, VC, alternative = "greater")
```

```
## Warning in wilcox.test.default(OJ, VC, alternative = "greater"): cannot
## compute exact p-value with ties
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  OJ and VC
## W = 575.5, p-value = 0.03225
## alternative hypothesis: true location shift is greater than 0
```

Lecture 13

Goodness-of-Fit Tests

A statistical test used to decide whether a data set came from a paricular distribution or not is known as a goodness-of-fit test since it decides whether the suggested distribution is a “good fit” for the data. Many goodness-of-fit tests exist, and here we will study the chi-square (or χ^2) tests. Here we look at chi^2 tests of two flavors: one deciding whether a categorical variable follows a particular distribution or not, and one deciding whether two categorical variables are independent or not.

I walked to 7-Eleven and bought a share-size bag of regular M&Ms, then counted how many M&Ms there were of each color. Below is the data set.

Red	Brown	Green	Yellow	Orange	Blue	Total
21	5	10	16	15	36	103

The official distribution of M&M candy colors in 1997 is listed below:

Red	Brown	Green	Yellow	Orange	Blue
.2	.3	.1	.2	.1	.1

Our question: Based off of our sample, should we believe that this is the distribution of colors in the bag? To be more precise, there are $K = 6$ possible categories (colors), we observe n_k candies for color k and there are $N = \sum_{k=1}^{K}$ n_k$ candies total. We have probabilities p_k of observing each of these colors for a randomly sampled candy and these probabilities are specified under the null hypothesis: denote the null-hypothesis probabilities as p_{k0}. We wish to decide between the null hypothesis

$$H_0 : p_1 = p_{10}, p_2 = p_{20}, \ldots, p_K = p_{K0}$$

and the alternative hypothesis which simply states that the null hypothesis is false (two or more of the null hypothesis probabilities are incorrect). To do this we compare the observed count of each category, n_k, against the expected count if the null hypothesis were true, given by Np_{k0}. We make these comparisons using the χ^2 statistic:

$$\sum_{k=1}^{K} \frac{(n_k - Np_{k0})^2}{Np_{k0}}.$$

If the null hypothesis is true, N is large, and $Np_{k0} > 10$, this statistic is well approximated by a χ^2 distribution with $K-1$ degrees of freedom. If the null hypothesis is false, we would expect to see at least one observed count far away from its expected count, causing that term in the statistic to be large and thus the overall statistic to be large. Thus large values of the χ^2 statistic are evidence against the null hypothesis and thus suggest rejecting it. This procedure together is the χ^2 test for goodness of fit, and the test can be performed in R using the function `chisq.test()`. We give this function first the observed counts, then the hypothesized distribution under then null hypothesis. It will then return the results of the χ^2 test.

```r
counts <- c(21, 5, 10, 16, 15, 36)
dist <- c(.2, .3, .1, .2, .1, .1)

chisq.test(counts, dist)
```

```
## Warning in chisq.test(counts, dist): Chi-squared approximation may be
## incorrect

##
##  Pearson's Chi-squared test
##
## data:  counts and dist
## X-squared = 12, df = 10, p-value = 0.2851
```

In this case the null hypothesis was not rejected, though the function warned that the assumptions (generally related to sample size) may not be satisfied and thus the asymptotic approach to computing the statistic (that is, using the χ^2 distribution) may not work. If this is in fact a concern then we can tell the function to use simulation methods to get a distribution that may be better for computing p-values. We can set the parameter B to tell the function how many simulations to do, the larger the better (but the default should be good).

```r
chisq.test(counts, dist, simulate.p.value = TRUE, B = 10000)
```

```
##
##  Pearson's Chi-squared test with simulated p-value (based on 10000
##  replicates)
```

```
##
## data:  counts and dist
## X-squared = 12, df = NA, p-value = 1
```

In any case it seems we can't reject the null hypothesis based on this data set (though another statistician reached a different conclusion).

If instead we wanted to test whether each color was equally likely or not, we can leave the second parameter unspecified, as the default null hypothesis is equal probability for all categories.

```
chisq.test(counts)
```

```
##
##  Chi-squared test for given probabilities
##
## data:  counts
## X-squared = 33.485, df = 5, p-value = 3.014e-06
```

In this case the null hypothesis was rejected; the true distribution of the data certainly does not appear to assign equal probability to all colors.

Let's now suppose we have two categorical variables and we want to determine whether those random variables are independent or not. In R we could refer to the `Titanic` data set, which tracks how many individuals died on the ship *Titanic* along with perhaps relevant information such as their age, sex, and class on the ship. We will have a two-way table tracking the number of people in each class and how many did or did not survive. We will get this table like so:

```
(class_survive_titanic <- apply(Titanic, c(1, 4), sum))
```

```
##       Survived
## Class   No Yes
##   1st  122 203
##   2nd  167 118
##   3rd  528 178
##   Crew 673 212
```

We wish to decide between the null hypothesis stating that the two variables are independent and the alternative hypothesis stating that the null hypothesis is false (they are not independent). In this case the observed counts will be the counts in each cell. The expected counts are estimated counts if the null hypothesis of independence were in fact true. Suppose that for variable A we have J possible categories and the probability of the observation belonging to category j is p_j, and for variable B there are K possible categories and the probability of observing category k is q_k. If the independence hypothesis is true then the probability that both category j and category k are observed is $p_j q_k$. We do not know these probabilities, though. Let n_{jk} be the number of observations falling in both categories j and k. Let $N_{j\cdot} = \sum_{k=1}^{K} n_{jk}$ and

$N_{\cdot k} = \sum_{j=1}^{J} n_{jk}$. Let $N = \sum_{j=1}^{J} \sum_{k=1}^{K} n_{jk}$. Then we would estimate p_j with $\hat{p}_j = \frac{N_{j\cdot}}{N}$ and $\hat{q}_k = \frac{N_{\cdot k}}{N}$. Our estimated count for the number of observations that belong both to category j for variable A and category k for variable B if the null hypothesis is true is $N\hat{p}_j\hat{q}_k = \frac{N_{j\cdot}N_{\cdot k}}{N} = o_{jk}$. Our test statistic will then be

$$\sum_{j=1}^{J} \sum_{k=1}^{K} \frac{(n_{jk} - o_{jk})^2}{o_{jk}}.$$

If the null hypothesis is true the approximate distribution of this statistic will be a χ^2 distribution with $(J-1)(K-1)$ degrees of freedom. As before, if the alternative is true, then we expect to see a cell count far away from what it should be if the null hypothesis were true, and the statistic will be large.

The function `chisq.test()` can also perform the test for independence. We can decide if class matters to surviving the *Titanic* disaster like so:

```r
chisq.test(class_survive_titanic)
```

```
##
##  Pearson's Chi-squared test
##
## data:  class_survive_titanic
## X-squared = 190.4, df = 3, p-value < 2.2e-16
```

In this case the null hypothesis is soundly rejected; class does seem to matter.

Lecture 14

Beyond Linear Models

In this section we will explore models that are not linear models, in that they are not linear in their parameters. Recall that linear models are models that take the form:

$$y_i = \beta_0 + \beta_1 x_{1i} + \cdots + \beta_k x_{ki} + \epsilon_i.$$

They either take that form exactly or could take that form after some transformation (commonly a log transformation). Before looking at nonlinear models, make sure that the model is not linear after a transformation. For instance, the model:

$$y_i = \beta_0 x_i^{\beta_1} \epsilon_i$$

is linear in the parameters after taking a log transform:

$$\log(y_i) = \log(\beta_0) + \beta_1 \log(x_i) + \log(\epsilon_i).$$

But the model below is not linear in the parameters:

$$y_i = \beta_0 e^{\beta_1 x_i} + \epsilon_i.$$

We will first talk about a particular nonlinear model: logistic regression. Then we will discuss estimating nonlinear models in general.

Logistic Regression

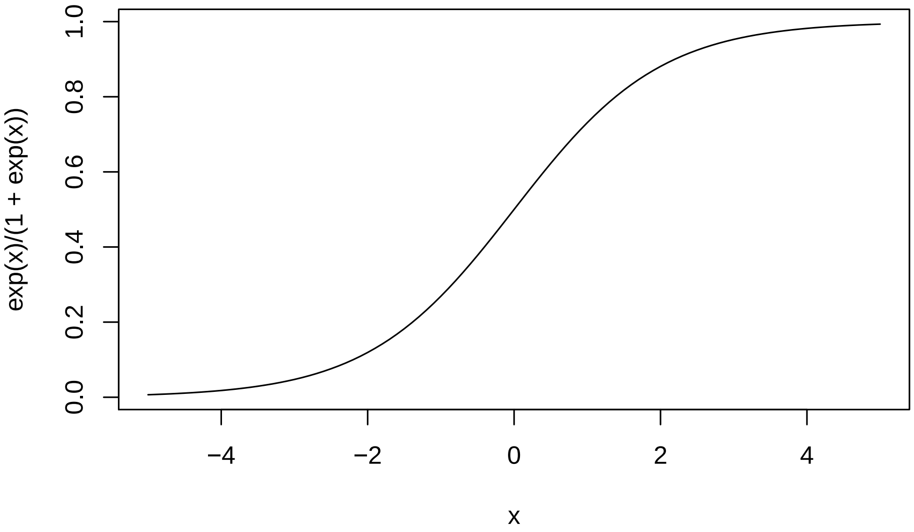
Suppose $y_i \in \{0, 1\}$; that is, y_i is a Bernoulli random variable. We would like to predict whether y_i is either 0 or 1 using a set of regressors $x_1, \ldots, x_k$. We could try using the linear model above, but unfortunately the error terms will not be *i.i.d.* and won't even look remotely Normal. Additionally, the resulting model will produce strange “predictions”; we would interpret the model as estimating the probability that $y_i = 1$ based on $x_1, \ldots, x_k$, but the model can give probabilities above 1 or below 0. These are undesirable properties that call for a different procedure.

Logisitic regression is a regression procedure that fixes these properties. Let π_i be the probability that $y_i = 1$. Logistic regression estimates the model

$$\pi_i = \frac{e^{\beta_0 + \beta_1 x_{1i} + \cdots + \beta_k x_{ki} + \epsilon_i}}{1 + e^{\beta_0 + \beta_1 x_{1i} + \cdots + \beta_k x_{ki} + \epsilon_i}} = g(\beta_0 + \beta_1 x_{1i} + \cdots + \beta_k x_{ki} + \epsilon_i).$$

The function $g(x) = \frac{e^x}{1+e^x}$ is known as the **logistic function**, and it only takes values between 0 and 1, and thus can produce probabilities.

```r
curve(exp(x)/(1 + exp(x)), from = -5, to = 5)
```



Another term for the model produced by logistic regression is **logit models**.

The predicted values from logistic regression are not 1 or 0 but the probability that the response variable is 1 or 0 given the values of the regressors. If you want to convert these probabilities into predictions, you would perhaps threshold these probabilities so that probabilities above, say, 0.5 will be predictions that $y_i = 1$ and all others predict $y_i = 0$.

The coefficients themselves are more difficult to interpret, but they can be interpreted. Recall that if the probability an event A occurs is p, then the *odds* that A occurs is $\frac{p}{1-p}$ to 1. It turns out that the logistic regression model can be rewritten as:

$$\ln\left(\frac{\pi_i}{1 - \pi_i}\right) = \beta_0 + \beta_1 x_{1i} + \cdots + \beta_k x_{ki} + \epsilon_i.$$

We call $\ln\left(\frac{\pi_i}{1-\pi_i}\right)$ the **log-odds** that $y_i = 1$, and so we are estimating a linear model to predict the log-odds that our response variable is 1. We could then say that a unit increase in x_{ji} predicts that the log-odds will increase by β_j, or that the odds will increase by a factor of e^{β_j}. (And for what it's worth, sometimes a logit model is accompanied by a linear model since linear model parameters are easily interpreted.)

Logit models must be estimated numerically, but the good news is that inference procedures we saw before, including marginal t-tests and AIC inference, still hold. In R, the function for estimating logit models is `glm()`, which can be understood as meaning "generalized linear model". `glm()` can actually estimate a lot of models, including models where we have specific assumptions about what the distribution of the response variables are (such as poisson or gamma), and much can be said about it. Here, though, I will only show how to use it for logistic regression.

If we want to estimate a logit model, our function call will resemble `glm(y ~ x, data = d, family = binomial)`. The function call is almost identical to `lm()`, but we have an additional parameter, `family`, that identifies the type of regression model we're estimating. By default, `family = binomial` performs logistic regression.

Let's demonstrate by building a logit model to predict whether an iris flower is a member of the versicolor species or not. We will use the sepal length, sepal width, petal length, and petal width for our predictions.

```
(fit <- glm(I(Species == "versicolor") ~ Sepal.Length + Sepal.Width +
            Petal.Length + Petal.Width, data = iris, family = binomial))
```

```
##
## Call:  glm(formula = I(Species == "versicolor") ~ Sepal.Length + Sepal.Width +
##     Petal.Length + Petal.Width, family = binomial, data = iris)
##
## Coefficients:
##  (Intercept)  Sepal.Length   Sepal.Width  Petal.Length   Petal.Width
##       7.3785       -0.2454       -2.7966        1.3136       -2.7783
##
## Degrees of Freedom: 149 Total (i.e. Null);  145 Residual
## Null Deviance:       191
## Residual Deviance: 145.1     AIC: 155.1
```

```
summary(fit)
```

```
##
## Call:
## glm(formula = I(Species == "versicolor") ~ Sepal.Length + Sepal.Width +
##     Petal.Length + Petal.Width, family = binomial, data = iris)
##
```

```
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1280  -0.7668  -0.3818   0.7866   2.1202
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)    7.3785     2.4993   2.952 0.003155 **
## Sepal.Length  -0.2454     0.6496  -0.378 0.705634
## Sepal.Width   -2.7966     0.7835  -3.569 0.000358 ***
## Petal.Length   1.3136     0.6838   1.921 0.054713 .
## Petal.Width   -2.7783     1.1731  -2.368 0.017868 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 190.95  on 149  degrees of freedom
## Residual deviance: 145.07  on 145  degrees of freedom
## AIC: 155.07
##
## Number of Fisher Scoring iterations: 5
```

```r
exp(coef(fit))
```

```
##  (Intercept) Sepal.Length  Sepal.Width Petal.Length  Petal.Width
## 1.601165e+03 7.824254e-01 6.101912e-02 3.719701e+00 6.214133e-02
```

```r
class(fit)
```

```
## [1] "glm" "lm"
```

`glm()` produces `glm`-class objects, which inherit from `lm`-class objects. So functions such as `predict()` or `coef()` work for these objects, and they have similar structures.

Nonlinear Models

A nonlinear model is not linear in its coefficients, and many of these models take the form:

$$y_i = f(x_{1i}, \ldots, x_{k1}; \beta_0, \beta_1, \ldots, \beta_r) + \epsilon_i.$$

(Note that r and k may be different.) I gave an example of a nonlinear model above, and this class of models is quite general. We still interpret $\beta_0, \ldots, \beta_r$ as parameters of the model and $f(\cdots; \beta_0, \ldots, \beta_r)$ as a function giving predicted values for y_i since the residuals ϵ_i are still assumed to be *i.i.d.* with mean 0. In fact the least-squares principle still holds since we want to pick $\hat{\beta}_0, \ldots, \hat{\beta}_r$ that minimize

$$SSE(\hat{\beta}_0, \dots, \hat{\beta}_r) = \sum_{i=1}^{n} \left(y_i - f(x_{1i}, \dots, x_{ki}; \hat{\beta}_0, \dots, \hat{\beta}_r)\right)^2.$$

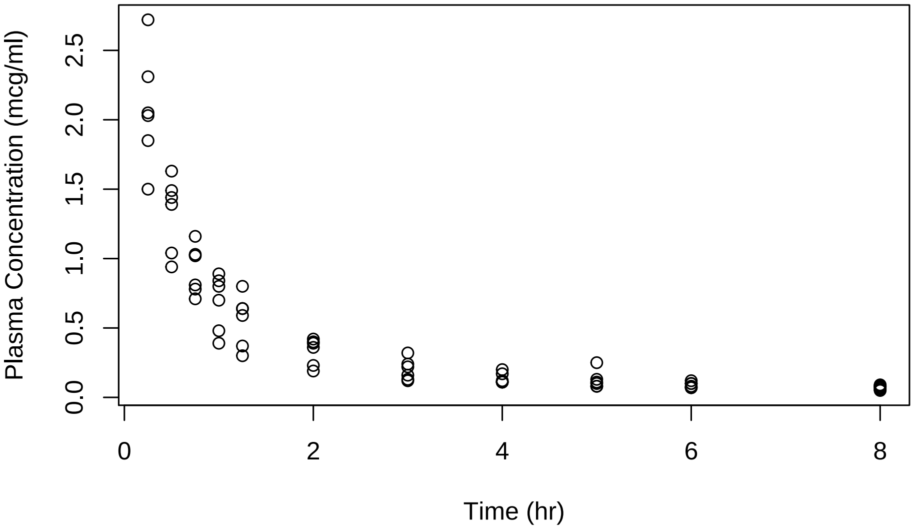
In the case of linear models, the coefficients of the model can be solved for explicitly, perhaps after invoking linear algebra. That is not the case in general and generally one must resort to numerical techniques to estimate nonlinear models. As for what form f itself takes, that's a domain-specific problem; there could be a physical or biological model that dictates the form of f, which is known up to the parameters (which we as statisticians then try to estimate).

Numerical optimization procedures/numerical solvers are iterative and generally need a set of starting parameters in order to work. This means that users of these methods need to make an educated guess as to what the true parameters are. Perhaps consider using plots to form guesses as to what the parameter values are, getting parameter values that appear to be "close" to fitting the data. (This is completely subjective, by the way.) Once you have a decent guess at the parameter values, you can then use the numerical procedures to get the proper fit.

The R function for nonlinear least squares estimation is `nls()`, whose function calls often take the form `nls(formula, data = d, start = v)`, where `d` is often a data frame and `v` is a vector of initial guesses for the parameter values. Due to the general nature of nonlinear least squares, `formula` generally resembles `y ~ f(variables, parameters)`, where `f` is a function. One could potentially write the nonlinear relationship directly into the formula, but I would advise defining the function `f` separately, outside of the formula.

Let's see an example. The data set `Indometh` contains the results of a study investigating the spread of the drug indomethacin (a drug for pain relief and reducing the swelling around joints) through the bodies of six test subjects. Below is a plot of the data.

```
plot(conc ~ time, data = Indometh, ylab = "Plasma Concentration (mcg/ml)",
     xlab = "Time (hr)")
```



This data does not look linear. We might consider a log transform of the variable `conc` for blood concentration, but that doesn't fix the problem either. We may prefer a nonlinear model to describe the relationship between these variables.

We will use instead a linearly combined bi-exponential model. Let C represent plasma concentration of the drug and t time. All other variables in the equation below are model parameters we will need to estimate.

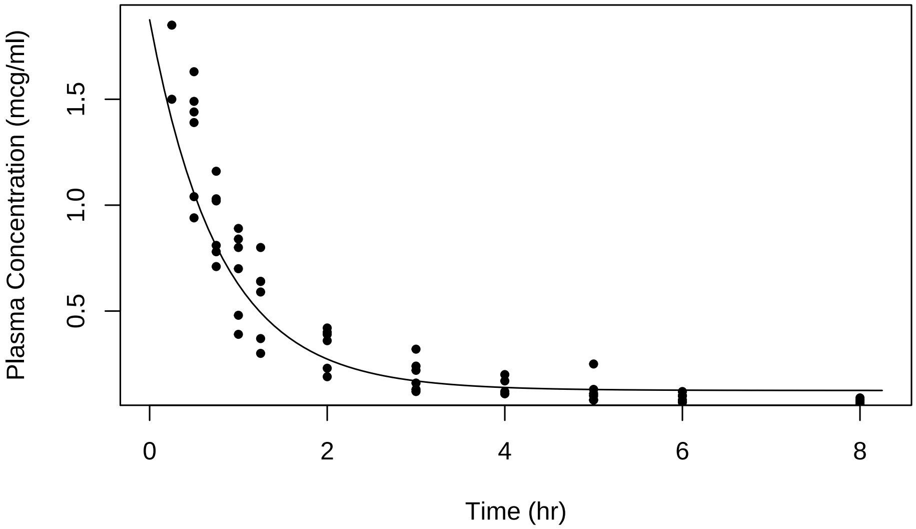
$$C_t = C_0 + a_1 e^{-b_1 t} + a_2 e^{-b_2 t}$$

Since we have different subjects i and measurement "errors" ϵ_{it}, we instead consider the following statistical model:

$$C_{it} = C_0 + a_1 e^{-b_1 t} + a_2 e^{-b_2 t} + \epsilon_{it}.$$

The R function below encapsulates the relationship between time and concentration.

```r
doubexpfunc <- function(t, C0, a1, a2, b1, b2) {
  C0 + a1 * exp(-b1 * t) + a2 * exp(-b2 * t)
}

curve(doubexpfunc(x, 0.125, 0.75, 1, 1.5, 1.1), from = 0, to = 8.25,
      xlab = "Time (hr)", ylab = "Plasma Concentration (mcg/ml)")
points(conc ~ time, data = Indometh, pch = 20)
```



Above is the function with a guess at the parameter values superimposed on the data. Is the guess good? No, but it's close enough for the numerical techniques.

We actually get an estimated fit using these initial guesses like so:

```
(fit <- nls(conc ~ doubexpfunc(time, C0, a1, a2, b1, b2), data = Indometh,
        start = c(C0 = 0.125, a1 = 0.75, a2 = 1, b1 = 1.5, b2 = 1.1)))
```

```
## Nonlinear regression model
##   model: conc ~ doubexpfunc(time, C0, a1, a2, b1, b2)
##    data: Indometh
##      C0      a1      a2      b1      b2
## 0.07393 2.53684 0.95687 3.01862 0.66572
##  residual sum-of-squares: 1.872
##
## Number of iterations to convergence: 9
## Achieved convergence tolerance: 7.851e-06
```

```
summary(fit)
```

```
##
## Formula: conc ~ doubexpfunc(time, C0, a1, a2, b1, b2)
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## C0  0.07393    0.06461   1.144   0.2570
## a1  2.53684    0.54544   4.651 1.82e-05 ***
## a2  0.95687    0.74799   1.279   0.2057
## b1  3.01862    1.40640   2.146   0.0358 *
## b2  0.66572    0.46825   1.422   0.1602
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1752 on 61 degrees of freedom
##
## Number of iterations to convergence: 9
## Achieved convergence tolerance: 7.851e-06
```

```
class(fit)
```

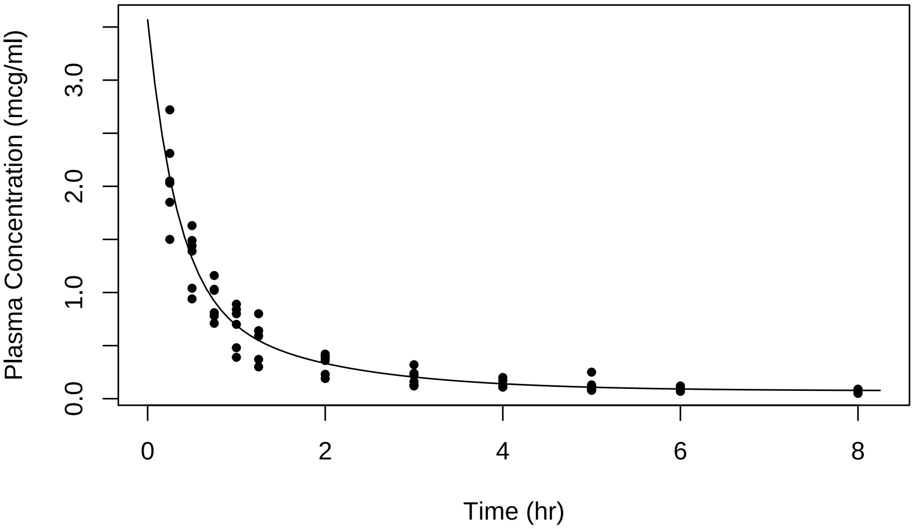
```
## [1] "nls"
```

```
coef(fit)
```

```
##         C0         a1         a2         b1         b2
## 0.07393449 2.53684071 0.95686744 3.01862352 0.66571921
```

A lot of the least-squares theory developed in the linear regression case carry over to nonlinear regression. For example, we can perform inference for model parameters using the *t*-test and assess the quality of the model fit with the AIC. Below is a plot of the resulting fit.

```
fitted_doubexpfunc <- function(fit) {
  C0 <- coef(fit)["C0"]
  a1 <- coef(fit)["a1"]
  a2 <- coef(fit)["a2"]
  b1 <- coef(fit)["b1"]
  b2 <- coef(fit)["b2"]
  function(t) {
    doubexpfunc(t, C0, a1, a2, b1, b2)
  }
}

fit_func <- fitted_doubexpfunc(fit)
curve(fit_func(x), from = 0, to = 8.25, xlab = "Time (hr)",
      ylab = "Plasma Concentration (mcg/ml)")
points(conc ~ time, data = Indometh, pch = 20)
```



All told the fit looks rather good. (Note that `predict()` can also be used for making predictions from the model.)

Be aware that nonlinear models are generally more complicated to estimate than linear models, if only because in addition to asking questions about the statistical quality about the model, practitioners should be sensitive to numerical issues as well. For example, one may need to decide on the maximum number of iterations for the solver, the desirable numerical tolerance, where to pick the starting values, what to do when the gradient matrix is near-singular, and so on. All told there are many more points of failure for nonlinear models that we will not discuss here.

Preface

These lecture notes were written by me to accompany John Verzani's *Using R for Introductory Statistics (2nd ed.)*, to be delivered in lectures teaching students how to program with R in the programming lab accompanying a lecture section focusing on the statistical methods themselves. This is for the second semester of a two-semester statistics course, so knowledge of basic statistical procedures (t-test, t confidence intervals for the mean, etc.) and basic R usage (creating variables, writing basic functions, working with lists and data frames, etc.) are assumed.

These notes are not intended to stand alone; I like Verzani's book and I believe that these notes should supplement it, not replace it. For those taking the programming lab for the University of Utah's Mathematics Department statistics courses, I would *insist* on reading Verzani's book in addition to these lecture notes. However, these notes could serve as a light weight introduction to R and statistical programming.

I hope that you find these notes useful, and wish you the best of luck.

Curtis Miller