MATH 3080 Lab Lecture 6

Curtis Miller

01/01/2020

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)
}
```

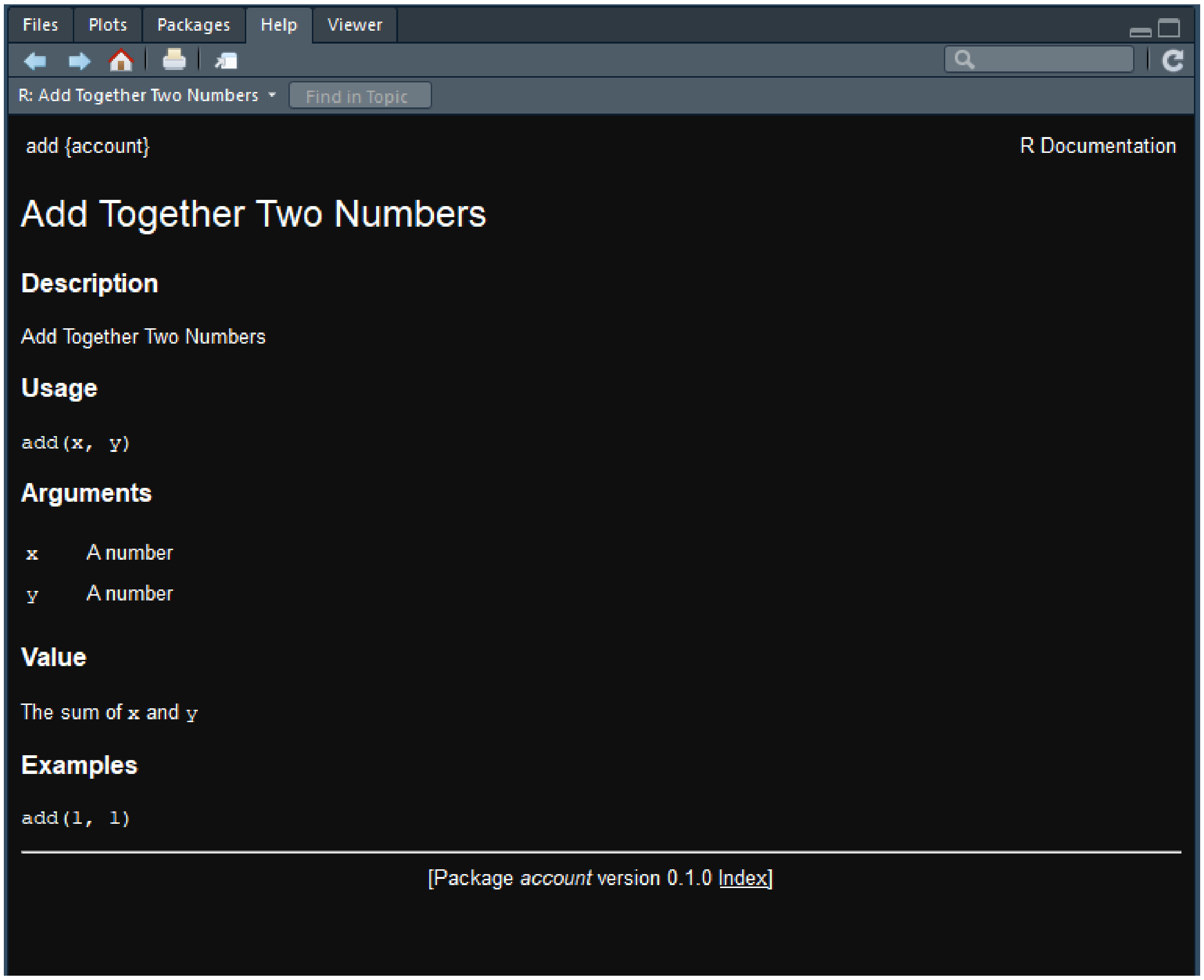
The necessary documentation was made. Now rebuild and install **account**. When we type `?add` at the command line we get documentation for the function `add()`:

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:

- `\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.



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 1: Resulting formatted documentation

`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
}
```

```r
#' @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)
`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)
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))
  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.
#'
```

```r
#' @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.
#'
#' @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
################################################################################
```

```r
#' 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))
}
```

```r
#' 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(...))
}
```

```r
#' 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.

```r
################################################################################
# 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
```

```r
################################################################################

#' 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.

```r
################################################################################
# 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"
#' )
```

```r
#' 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`.

```r
################################################################################
# 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:

```r
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
################################################################################
# Curtis Miller
# 2020-01-10
################################################################################
# Tests for transaction-class object helper functions
################################################################################

context("transaction-class object helper functions")

################################################################################
# SETUP
################################################################################
```

```r
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.

```r
################################################################################
# test-account.R
################################################################################
# Curtis Miller
# 2020-01-10
################################################################################
# Tests for account objects and their methods
################################################################################

context("account objects")

################################################################################
# SETUP
################################################################################

library(account)
```

```r
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
################################################################################
```

```r
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 `:::`.

```r
################################################################################
# test-accountHelpers.R
```

```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.

```r
################################################################################
# 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")
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.

```r
################################################################################
# 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))
```

```r
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.

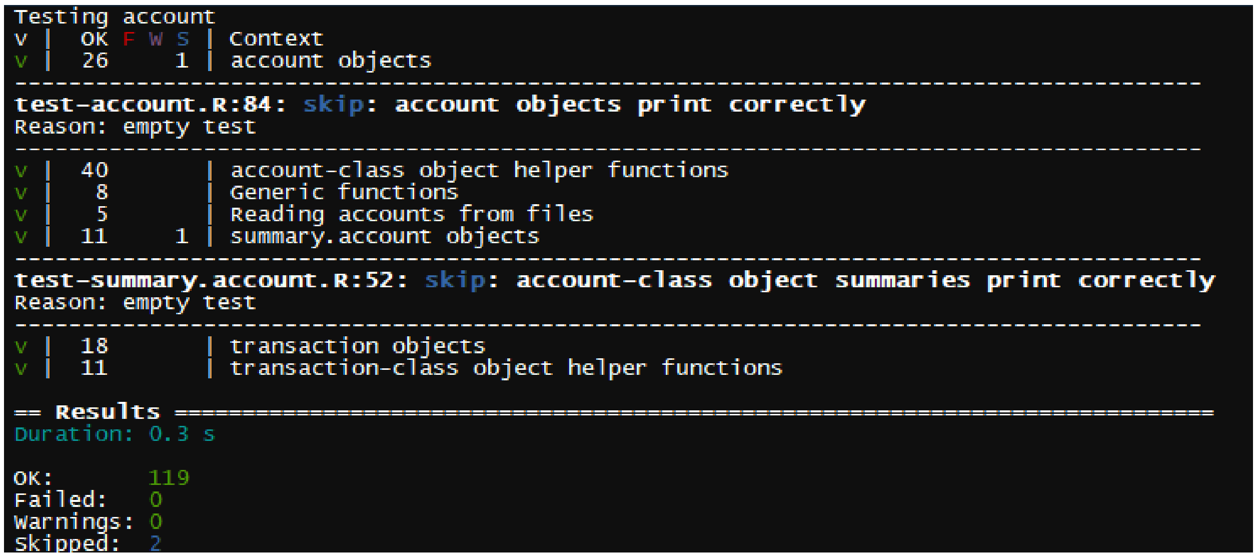
```r
################################################################################
# test-read_csv_account.R
# Curtis Miller
# 2020-01-13
################################################################################
# Testing functions for reading account objects
################################################################################

context("Reading accounts from files")

################################################################################
# SETUP
################################################################################

library(account)

input_str1 <- "date,amount,memo
```

```r
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 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.



```
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 2: Testing